**算法分析与设计实验报告**

**第 4 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 杨杰 | 学号 | 201908010705 | | 班级 | 计科1907 |
| 时间 | 6.2 | 地点 | 软件大楼 | | | |
| 实验名称 | 用分支限界法求解TSP问题 | | | | | |
| 实验目的 | 通过上机实验，要求掌握用分支限界法求解TSP问题的问题描述、算法设计思想、程序设计。 | | | | | |
| 实验原理 | 旅行售货员问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图G中找出费用最小的周游路线。路线是一个带权图。图中各边的费用（权）为正数。图的一条周游路线是包括V中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。  算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界lcost值是优先队列的优先级。接着算法计算出图中每个顶点的最小费用出边并用minout记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的minout作算法初始化。  算法的while循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：  1、首先考虑s=n-2的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。  2、当s<n-2时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是x[0:s]，其可行儿子结点是从剩余顶点x[s+1:n-1]中选取的顶点x[i]，且(x[s]，x[i])是所给有向图G中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀(x[0:s]，x[i])的费用cc和相应的下界lcost。当lcost<bestc时，将这个可行儿子结点插入到活结点优先队列中。  算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。当s=n-1时，已找到的回路前缀是x[0:n-1]，它已包含图G的所有n个顶点。因此，当s=n-1时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于cc和lcost的值。剩余的活结点的lcost值不小于已找到的回路的费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。  算法结束时返回找到的最小费用，相应的最优解由数组v给出。 | | | | | |
| 实验步骤 | 1. 对根节点进行初始化 2. 若当前扩展节点是叶节点的父节点，根据约束函数和限界函数判断是否扩展，若扩展，加入活结点优先队列 3. 若当前扩展节点是其他节点，扩展所有满足约束函数和限界函数的儿子节点，并加入活结点优先队列 4. 从活结点优先队列中取出优先级最高的节点并扩展 5. 重复2-4步直到当前扩展节点为叶节点，构造解向量 | | | | | |
| 关键代码 | *//解旅行员售货问题的优先队列式分支限界法*  template <class Type>  Type Traveling<Type>::BBTSP(int v[])  {      priority\_queue<MinHeapNode<Type>> Q;      Type \*MinOut = new Type[n + 1];  *//计算MinOut[i] = 顶点i的最小出边费用*      Type MinSum = 0;*//最小出边费用和*      for (int i = 1; i <= n; i++)      {          Type Min = NoEdge;          for (int j = 1; j <= n; j++)          {              if (a[i][j] != NoEdge && (a[i][j] < Min || Min == NoEdge))              {                  Min = a[i][j];              }          }          if (Min == NoEdge)          {              return NoEdge;*//无回路*          }          MinOut[i] = Min;          MinSum += Min;      }  *//初始化*      MinHeapNode<Type> E;      E.x = new int[n];      for (int i = 0; i < n; i++)      {          E.x[i] = i + 1;      }      E.s = 0;*//根节点到当前节点路径为x[0:s]*      E.cc = 0;*//当前费用*      E.rcost = MinSum;*//最小出边费用和*      Type bestc = NoEdge;  *//搜索排列空间树*      while (E.s < n - 1)*//非叶结点*      {          if (E.s == n - 2)*//当前扩展节点是叶节点的父节点*          {  *//再加2条边构成回路*  *//所构成回路是否优于当前最优解*              if (a[E.x[n - 2]][E.x[n - 1]] != NoEdge && a[E.x[n - 1]][1] != NoEdge && (E.cc + a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1] < bestc || bestc == NoEdge))              {  *//费用更小的回路*                  bestc = E.cc + a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1];                  E.cc = bestc;                  E.lcost = bestc;                  E.s++;                  Q.push(E);              }              else              {                  delete[] E.x;*//舍弃扩展节点*              }          }          else*//产生当前扩展节点的儿子节点*          {              for (int i = E.s + 1; i < n; i++)              {                  if (a[E.x[E.s]][E.x[i]] != NoEdge)                  {  *//可行儿子节点*                      Type cc = E.cc + a[E.x[E.s]][E.x[i]];                      Type rcost = E.rcost - MinOut[E.x[E.s]];                      Type b = cc + rcost;*//下界*                      if (b < bestc || bestc == NoEdge)                      {  *//子树可能含有最优解*  *//节点插入最小堆*                          MinHeapNode<Type> N;                          N.x = new int[n];                          for (int j = 0; j < n; j++)                          {                              N.x[j] = E.x[j];                          }                          N.x[E.s + 1] = E.x[i];                          N.x[i] = E.x[E.s + 1];                          N.cc = cc;                          N.s = E.s + 1;                          N.lcost = b;                          N.rcost = rcost;                          Q.push(N);                      }                  }              }              delete[] E.x;*//舍弃扩展节点*          }          if (Q.empty())          {              break;          }          E = Q.top();*//取下一扩展节点*          Q.pop();      }      if (bestc == NoEdge)      {          return NoEdge;*//无回路*      }  *//将最优解复制到v[1:n]*      for (int i = 0; i < n; i++)      {          v[i + 1] = E.x[i];      }      while (true)*//释放最小堆中所有节点*      {          delete[] E.x;          if (Q.empty())          {              break;          }          E = Q.top();*//取下一扩展节点*          Q.pop();      }      return bestc;  } | | | | | |
| 算法复杂度分析 | 分支限界法解TSP所用的时间是少于回溯法的，毕竟前者设计上便是以空间换时间。回溯法的时间复杂度是O(n!)，而分支限界法的时间复杂度是O (n2×2n )。  分支限界法解TSP的空间复杂度是O(n!)。 | | | | | |
| 测试结果  （含运行时间） | **小规模数据**    **中规模数据**    **大规模数据**        当n=25时内存就已经不够了。（我的电脑内存太小……） | | | | | |
| 实验心得 | 之前在这里犯了一个很低级的错误，就是重复定义了E这个变量，导致结果怎么都不对，所以还是要更加细心才是。  还要注意关于queue和priority\_queue取队首元素的区别，queue是q.front()，而priority\_queue是q.top()。  经过这次实验，我对于用分支限界法求解TSP问题的相关代码已基本熟悉，算法知识得到了复习与巩固。在写代码与调试的过程中，在解决问题过程中，丰富了个人编程的经历和经验，提高了个人解决问题的能力。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

*//分支限界法求解TSP问题*

#include <iostream>

#include <fstream>

#include <windows.h>

#include <queue>

using namespace std;

ifstream infile("TSP\_in.txt", ios::in);

ofstream outfile("TSP\_out.txt", ios::out);

int N;

template <class Type>

class Traveling

{

    friend int main();

public:

    Type BBTSP(int v[]);

private:

    int n;*//图G的顶点数*

    Type \*\*a,*//图G的邻接矩阵*

        NoEdge,*//图G的无边标识*

        cc,*//当前费用*

        bestc;*//当前最小费用*

};

template <class Type>

class MinHeapNode

{

    friend Traveling<Type>;

    friend bool operator<(MinHeapNode a, MinHeapNode b) { return a.lcost > b.lcost; };*//以lcost小为优先级*

private:

    Type lcost,*//子树费用的下界*

        cc,*//当前费用*

        rcost;*//x[s:nNoEdge]中顶点最小出边费用和*

    int s,*//根节点到当前节点的路径为x[0:s]*

        \*x;*//需要进一步搜索的顶点是x[s+1:nNoEdge]*

};

*//解旅行员售货问题的优先队列式分支限界法*

template <class Type>

Type Traveling<Type>::BBTSP(int v[])

{

    priority\_queue<MinHeapNode<Type>> Q;

    Type \*MinOut = new Type[n + 1];

*//计算MinOut[i] = 顶点i的最小出边费用*

    Type MinSum = 0;*//最小出边费用和*

    for (int i = 1; i <= n; i++)

    {

        Type Min = NoEdge;

        for (int j = 1; j <= n; j++)

        {

            if (a[i][j] != NoEdge && (a[i][j] < Min || Min == NoEdge))

            {

                Min = a[i][j];

            }

        }

        if (Min == NoEdge)

        {

            return NoEdge;*//无回路*

        }

        MinOut[i] = Min;

        MinSum += Min;

    }

*//初始化*

    MinHeapNode<Type> E;

    E.x = new int[n];

    for (int i = 0; i < n; i++)

    {

        E.x[i] = i + 1;

    }

    E.s = 0;*//根节点到当前节点路径为x[0:s]*

    E.cc = 0;*//当前费用*

    E.rcost = MinSum;*//最小出边费用和*

    Type bestc = NoEdge;

*//搜索排列空间树*

    while (E.s < n - 1)*//非叶结点*

    {

        if (E.s == n - 2)*//当前扩展节点是叶节点的父节点*

        {

*//再加2条边构成回路*

*//所构成回路是否优于当前最优解*

            if (a[E.x[n - 2]][E.x[n - 1]] != NoEdge && a[E.x[n - 1]][1] != NoEdge && (E.cc + a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1] < bestc || bestc == NoEdge))

            {

*//费用更小的回路*

                bestc = E.cc + a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1];

                E.cc = bestc;

                E.lcost = bestc;

                E.s++;

                Q.push(E);

            }

            else

            {

                delete[] E.x;*//舍弃扩展节点*

            }

        }

        else*//产生当前扩展节点的儿子节点*

        {

            for (int i = E.s + 1; i < n; i++)

            {

                if (a[E.x[E.s]][E.x[i]] != NoEdge)

                {

*//可行儿子节点*

                    Type cc = E.cc + a[E.x[E.s]][E.x[i]];

                    Type rcost = E.rcost - MinOut[E.x[E.s]];

                    Type b = cc + rcost;*//下界*

                    if (b < bestc || bestc == NoEdge)

                    {

*//子树可能含有最优解*

*//节点插入最小堆*

                        MinHeapNode<Type> N;

                        N.x = new int[n];

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

                        {

                            N.x[j] = E.x[j];

                        }

                        N.x[E.s + 1] = E.x[i];

                        N.x[i] = E.x[E.s + 1];

                        N.cc = cc;

                        N.s = E.s + 1;

                        N.lcost = b;

                        N.rcost = rcost;

                        Q.push(N);

                    }

                }

            }

            delete[] E.x;*//舍弃扩展节点*

        }

        if (Q.empty())

        {

            break;

        }

        E = Q.top();*//取下一扩展节点*

        Q.pop();

    }

    if (bestc == NoEdge)

    {

        return NoEdge;*//无回路*

    }

*//将最优解复制到v[1:n]*

    for (int i = 0; i < n; i++)

    {

        v[i + 1] = E.x[i];

    }

    while (true)*//释放最小堆中所有节点*

    {

        delete[] E.x;

        if (Q.empty())

        {

            break;

        }

        E = Q.top();*//取下一扩展节点*

        Q.pop();

    }

    return bestc;

}

Traveling<int> t;

int main()

{

    infile >> N;

    int \*bestx = new int[N + 1];

    outfile << "图的顶点个数 n=" << N << endl;

    int \*\*a = new int \*[N + 1];

    for (int i = 1; i <= N; i++)

    {

        a[i] = new int[N + 1];

    }

    outfile << "图的邻接矩阵为:" << endl;

    for (int i = 1; i <= N; i++)

    {

        for (int j = 1; j <= N; j++)

        {

            infile >> a[i][j];

            outfile << a[i][j] << " ";

        }

        outfile << endl;

    }

    t.a = a;

    t.n = N;

    double time = 0;

    LARGE\_INTEGER nFreq, nBeginTime, nEndTime;

    QueryPerformanceFrequency(&nFreq);

    QueryPerformanceCounter(&nBeginTime);*//开始计时*

    int dis = t.BBTSP(bestx);

    QueryPerformanceCounter(&nEndTime);*//停止计时*

    time = (double)(nEndTime.QuadPart - nBeginTime.QuadPart) / (double)nFreq.QuadPart;*//计算程序执行时间单位为s*

    outfile << "最短回路的长为：" << dis << endl;

    outfile << "最短回路为：" << endl;

    for (int i = 1; i <= N; i++)

    {

        outfile << bestx[i] << "-->";

    }

    outfile << bestx[1] << endl;

    outfile << "程序耗时：" << time \* 1000 << "ms";

    for (int i = 1; i <= N; i++)

    {

        delete[] a[i];

    }

    delete[] a;

    delete[] bestx;

    return 0;

}

*//TSP测试数据生成器*

#include <iostream>

#include <fstream>

#include <time.h>

#define random(a, b) (rand() % (b - a + 1) + a)

using namespace std;

int main()

{

    ofstream outfile("TSP\_in.txt", ios::out);

    srand((int)time(NULL));

    int n;

    cin >> n;

    outfile << n << endl;

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

            if (i == j)

                outfile << 0 << " ";

            else

                outfile << random(1, 50) << " ";

        outfile << endl;

    }

    outfile.close();

    return 0;

}