《算法分析与设计》

实 验 报 告

**学 号：2022112566**

**姓 名：倪阳**

**年 级：2022级**

**专 业：计算机科学与技术**

**院 系：计算机与人工智能学院**

**二 0 二四年五月**

**1. 实验目的**

（1） 实验比较数据输入规模对算法执行时间的影响。

（2） 实验比较不同算法对问题求解时间的影响。

（3） 掌握算法时间效率的分析方法。

**2. 实验任务**

（1） 分析函数Tsp在下面的样例输入下的执行过程及主要变量的变化情

况。

4 8 6 5 3 2 10 8 7 3

（2） 采用priority\_queue对上述程序进行修改，实现程序中对堆的操

作。

（3） 修改该程序使其可以输入任意多个城市，并根据城市多少动态分配

数组C的大小，同时求解出最短回路。

**3. 实验环境**

**3.1 硬件环境**

（1） 计算机：HUAWEI MateBook 14s

（2） CPU: 12th Gen Intel(R) Core(TM) i712700H 2.30 GHz

（3） RAM：16GB

**3.2 软件环境**

（1） 操作系统：Windows 11 专业版

（2） 开发工具：Visual Studio 2022

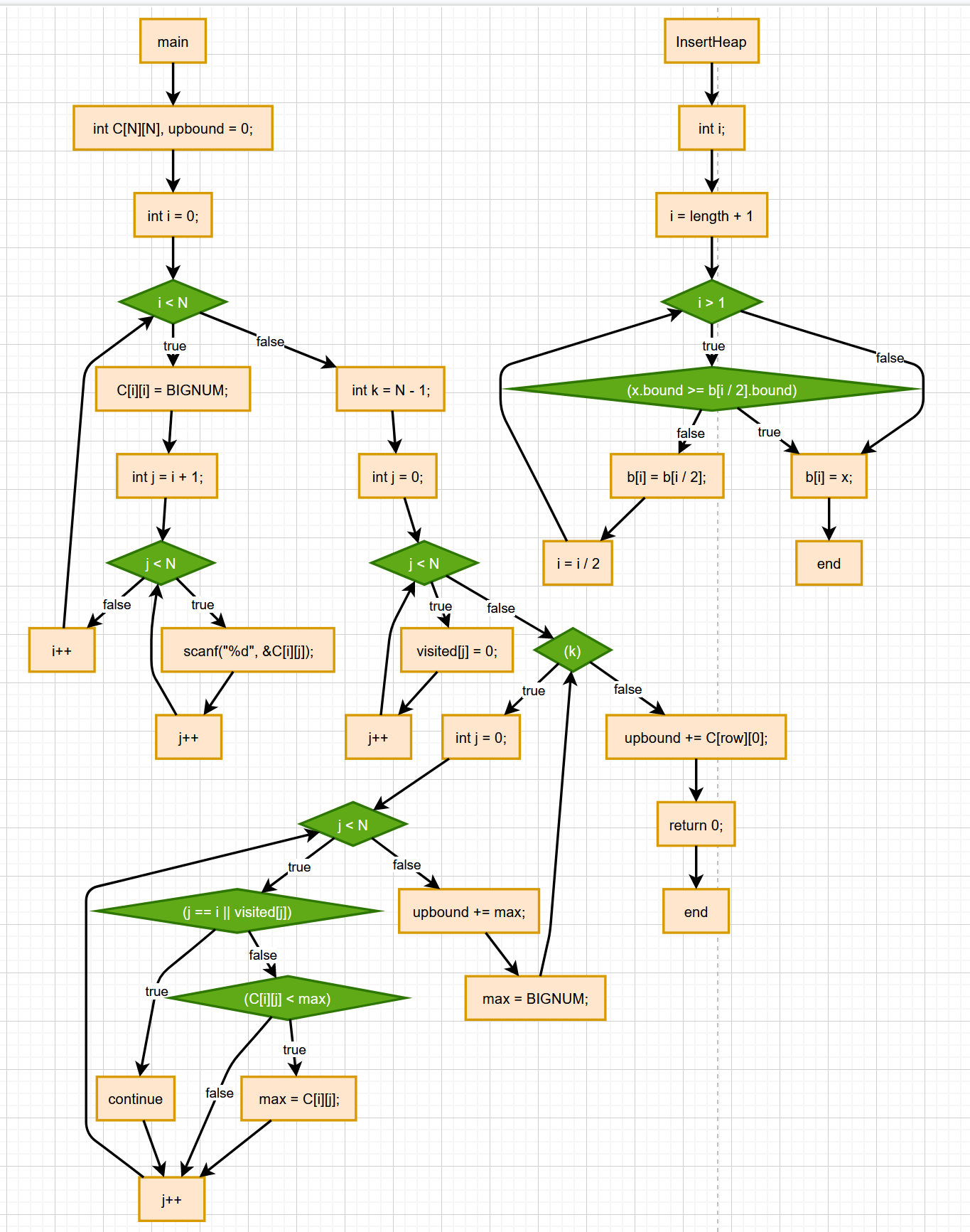
**4. 实验步骤及结果**

**4.1 实验预习**

**4.1.1 该问题的自然语言描述法**

这个问题是旅行商问题（TSP），目标是找到一条最短路径，使得旅行商可以经过每个城市一次，然后返回起点城市。我们可以使用优先队列（priority\_queue）来实现对堆的操作，以求解TSP问题。优先队列可以根据每个节点的上界（bound）来动态调整搜索顺序，从而更快地找到最优解。在每一步中，我们选择一个节点进行扩展，并计算其子节点的上界，然后将子节点加入优先队列中。重复这个过程，直到找到最优解或者队列为空为止。

**4.1.2 程序流程描述**



**4.1.3 程序代码**

#include "stdio.h"

#include "malloc.h"

#include "math.h"

#include<iostream>

using namespace std;

#define N 5

#define BIGNUM 1e9

struct HeapNode {

int v; //路径长度

int level; //所在层

int bound; //上界

int route[N + 1]; //路径，解向量的各个分量

};

void InsertHeap(HeapNode b[], HeapNode x, int &length);

void DeleteHeap(HeapNode b[], int &length);

void Tsp(int C[N][N], int upbound);

int RowMinExcept(int C[N][N], int row, int col);

int SiSum(int C[N][N], HeapNode temp);

int RowMinTwo(int C[N][N], int row);

bool Contain(HeapNode temp, int node);

int SjSum(int C[N][N], HeapNode temp);

HeapNode heap[100];

int heapLength = 0;

int level = 0;

int main() {

int C[N][N], upbound = 0;

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

C[i][i] = BIGNUM;

for (int j = i + 1; j < N; j++) {

scanf("%d", &C[i][j]);

C[j][i] = C[i][j];

}

}

int k = N - 1;

int i = 0;

int max = BIGNUM;

int visited[N];

int row;

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

visited[j] = 0;

while (k) {

for (int j = 0; j < N; j++) {

if (j == i || visited[j])

continue;

if (C[i][j] < max) {

max = C[i][j];

row = j;

}

}

upbound += max;

visited[i] = 1;

i = row;

k--;

max = BIGNUM;

}

upbound += C[row][0];

Tsp(C, upbound);

return 0;

}

void InsertHeap(HeapNode b[], HeapNode x, int &length) {

int i;

for (i = length + 1; i > 1; i = i / 2) {

if (x.bound >= b[i / 2].bound)

break;

else

b[i] = b[i / 2];

}

b[i] = x;

length += 1;

}

void DeleteHeap(HeapNode b[], int &length) {

int i;

b[1] = b[length];

HeapNode temp = b[1];

b[length].bound = BIGNUM;

for (i = 2; i < length; i \*= 2) {

if (b[i].bound > b[i + 1].bound)

i++;

if (temp.bound <= b[i].bound)

break;

b[i / 2] = b[i];

i = i / 2;

b[i] = temp;

length -= 1;

}

}

int RowMinExcept(int C[N][N], int row, int col) {

int min = BIGNUM;

for (int j = 0; j < N; j++) {

if (j == col)

continue;

if (C[row][j] < min)

min = C[row][j];

}

return min;

}

int SiSum(int C[N][N], HeapNode temp) {

int level = temp.level;

int sum = RowMinExcept(C, temp.route[0] - 1, temp.route[1] - 1)

+ RowMinExcept(C, temp.route[level] - 1, temp.route[level - 1] - 1);

return sum;

}

int RowMinTwo(int C[N][N], int row) {

int minOne = BIGNUM;

int minTwo = 0;

int col;

for (int j = 0; j < N; j++) {

if (C[row][j] < minOne) {

minOne = C[row][j];

col = j;

}

}

minTwo = RowMinExcept(C, row, col);

return minOne + minTwo;

}

bool Contain(HeapNode temp, int node) {

int level = temp.level; //获得当前已处理的结点

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

if (temp.route[i] == node)

return true;

}

return false;

}

int SjSum(int C[N][N], HeapNode temp) {

int sum = 0;

for (int i = 1; i <= N; i++) {

if (!Contain(temp, i)) {

sum = sum + RowMinTwo(C, i - 1);

}

}

return sum;

}

void Tsp(int C[N][N], int upbound) {

HeapNode rootNode;

rootNode.v = 0;

rootNode.level = 0;

rootNode.route[0] = 1;

rootNode.bound = SjSum(C, rootNode) / 2.0;

InsertHeap(heap, rootNode, heapLength); //在heap堆中插入根节点

while (heapLength > 0) {

HeapNode temp; //temp指向堆顶元素

temp = heap[1];

level = temp.level;

cout<<"v,level,bound = ";

cout<<temp.v<<','<<temp.level<<','<<temp.bound<<endl;

DeleteHeap(heap, heapLength);

if ((level == N)) {

for (int i = 0; i <= N; i++) { //输出该结点所对应的路径

printf("%d ", temp.route[i]);

}

printf("\n%d", temp.v);

return;

} else {

if ((level == N - 1) && C[temp.route[level] - 1][temp.route[0] - 1] != BIGNUM) {

temp.route[level + 1] = temp.route[0];

float lb = temp.v + C[temp.route[level] - 1][temp.route[0] - 1];

if (lb < upbound) {

HeapNode node;

node.v = lb;

node.bound = lb;

for (int i = 0; i <= level + 1; i++)

node.route[i] = temp.route[i];

node.level = level + 1;

InsertHeap(heap, node, heapLength);

upbound = lb;

}

} else {

//在若干个城市中选择

for (int j = 1; j <= N; j++) {

if ((C[temp.route[level] - 1][j - 1] != BIGNUM) && !Contain(temp, j)) {

temp.route[level + 1] = j;

temp.level = level + 1;

float lb = ((temp.v + C[temp.route[level] - 1][j - 1]) \* 2 + SiSum(C, temp) + SjSum(C, temp)) / 2.0;

if (lb <= upbound) {

HeapNode node;

node.v = temp.v + C[temp.route[level] - 1][j - 1];

node.bound = lb;

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

node.route[i] = temp.route[i];

node.route[level + 1] = j;

node.level = level + 1;

InsertHeap(heap, node, heapLength);

}

}

}

}

}

}

}

**4.1.4 分析函数Tsp在下面的样例输入下的执行过程及主要变量的变化情况**

4 8 6 5 3 2 10 8 7 3

v,level,bound = 0,0,14

v,level,bound = 5,1,18

v,level,bound = 8,2,18

v,level,bound = 10,3,18

v,level,bound = 13,4,18

v,level,bound = 21,5,21

1 5 4 2 3 1

21

**4.1.5 采用priority\_queue对上述程序进行修改，实现程序中对堆的操作**

#include "stdio.h"

#include "malloc.h"

#include "math.h"

#include <queue>

#define N 5

#define BIGNUM 1e9

struct HeapNode {

int v; //路径长度

int level; //所在层

int bound; //上界

int route[N + 1]; //路径，解向量的各个分量

bool operator<(const HeapNode& other) const {

return bound > other.bound; // 重载 < 运算符，使得 priority\_queue 可以按照 bound 从小到大排序

}

};

void Tsp(int C[N][N], int upbound);

int RowMinExcept(int C[N][N], int row, int col);

int SiSum(int C[N][N], HeapNode temp);

int RowMinTwo(int C[N][N], int row);

bool Contain(HeapNode temp, int node);

int SjSum(int C[N][N], HeapNode temp);

int level = 0;

int main() {

int C[N][N], upbound = 0;

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

C[i][i] = BIGNUM;

for (int j = i + 1; j < N; j++) {

scanf("%d", &C[i][j]);

C[j][i] = C[i][j];

}

}

int k = N - 1;

int i = 0;

int max = BIGNUM;

int visited[N];

int row;

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

visited[j] = 0;

while (k) {

for (int j = 0; j < N; j++) {

if (j == i || visited[j])

continue;

if (C[i][j] < max) {

max = C[i][j];

row = j;

}

}

upbound += max;

visited[i] = 1;

i = row;

k--;

max = BIGNUM;

}

upbound += C[row][0];

Tsp(C, upbound);

return 0;

}

int RowMinExcept(int C[N][N], int row, int col) {

int min = BIGNUM;

for (int j = 0; j < N; j++) {

if (j == col)

continue;

if (C[row][j] < min)

min = C[row][j];

}

return min;

}

int SiSum(int C[N][N], HeapNode temp) {

int level = temp.level;

int sum = RowMinExcept(C, temp.route[0] - 1, temp.route[1] - 1)

+ RowMinExcept(C, temp.route[level] - 1, temp.route[level - 1] - 1);

return sum;

}

int RowMinTwo(int C[N][N], int row) {

int minOne = BIGNUM;

int minTwo = 0;

int col;

for (int j = 0; j < N; j++) {

if (C[row][j] < minOne) {

minOne = C[row][j];

col = j;

}

}

minTwo = RowMinExcept(C, row, col);

return minOne + minTwo;

}

bool Contain(HeapNode temp, int node) {

int level = temp.level; //获得当前已处理的结点

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

if (temp.route[i] == node)

return true;

}

return false;

}

int SjSum(int C[N][N], HeapNode temp) {

int sum = 0;

for (int i = 1; i <= N; i++) {

if (!Contain(temp, i)) {

sum = sum + RowMinTwo(C, i - 1);

}

}

return sum;

}

void Tsp(int C[N][N], int upbound) {

std::priority\_queue<HeapNode> pq;

HeapNode rootNode;

rootNode.v = 0;

rootNode.level = 0;

rootNode.route[0] = 1;

rootNode.bound = SjSum(C, rootNode) / 2.0;

pq.push(rootNode); //在 priority\_queue 中插入根节点

while (!pq.empty()) {

HeapNode temp; //temp指向堆顶元素

temp = pq.top();

pq.pop();

level = temp.level;

if ((level == N)) {

for (int i = 0; i <= N; i++) { //输出该结点所对应的路径

printf("%d ", temp.route[i]);

}

printf("\n%d", temp.v);

return;

} else {

if ((level == N - 1) && C[temp.route[level] - 1][temp.route[0] - 1] != BIGNUM) {

temp.route[level + 1] = temp.route[0];

float lb = temp.v + C[temp.route[level] - 1][temp.route[0] - 1];

if (lb < upbound) {

HeapNode node;

node.v = lb;

node.bound = lb;

for (int i = 0; i <= level + 1; i++)

node.route[i] = temp.route[i];

node.level = level + 1;

pq.push(node);

upbound = lb;

}

} else {

//在若干个城市中选择

for (int j = 1; j <= N; j++) {

if ((C[temp.route[level] - 1][j - 1] != BIGNUM) && !Contain(temp, j)) {

temp.route[level + 1] = j;

temp.level = level + 1;

float lb = ((temp.v + C[temp.route[level] - 1][j - 1]) \* 2 + SiSum(C, temp) + SjSum(C, temp)) / 2.0;

if (lb <= upbound) {

HeapNode node;

node.v = temp.v + C[temp.route[level] - 1][j - 1];

node.bound = lb;

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

node.route[i] = temp.route[i];

node.route[level + 1] = j;

node.level = level + 1;

pq.push(node);

}

}

}

}

}

}

}

**4.1.6修改该程序使其可以输入任意多个城市，并根据城市多少动态分配数组C的大小，同时求解出最短回路。**

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#define BIGNUM std::numeric\_limits<int>::max()

struct HeapNode {

int v; //路径长度

int level; //所在层

int bound; //上界

std::vector<int> route; //路径，解向量的各个分量

bool operator<(const HeapNode& other) const {

return bound > other.bound; // 重载 < 运算符，使得 priority\_queue 可以按照 bound 从小到大排序

}

};

int RowMinExcept(const std::vector<std::vector<int>>& C, int row, int col) {

int min = BIGNUM;

for (int j = 0; j < C.size(); j++) {

if (j == col)

continue;

if (C[row][j] < min)

min = C[row][j];

}

return min;

}

int SiSum(const std::vector<std::vector<int>>& C, HeapNode temp) {

int level = temp.level;

int sum = RowMinExcept(C, temp.route[0] - 1, temp.route[1] - 1)

+ RowMinExcept(C, temp.route[level] - 1, temp.route[level - 1] - 1);

return sum;

}

int RowMinTwo(const std::vector<std::vector<int>>& C, int row) {

int minOne = BIGNUM;

int minTwo = 0;

int col;

for (int j = 0; j < C.size(); j++) {

if (C[row][j] < minOne) {

minOne = C[row][j];

col = j;

}

}

minTwo = RowMinExcept(C, row, col);

return minOne + minTwo;

}

bool Contain(const HeapNode& temp, int node) {

int level = temp.level; //获得当前已处理的结点

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

if (temp.route[i] == node)

return true;

}

return false;

}

int SjSum(const std::vector<std::vector<int>>& C, HeapNode temp) {

int sum = 0;

for (int i = 1; i <= C.size(); i++) {

if (!Contain(temp, i)) {

sum = sum + RowMinTwo(C, i - 1);

}

}

return sum;

}

void Tsp(const std::vector<std::vector<int>>& C, int upbound) {

std::priority\_queue<HeapNode> pq;

HeapNode rootNode;

rootNode.v = 0;

rootNode.level = 0;

rootNode.route.push\_back(1);

rootNode.bound = SjSum(C, rootNode) / 2.0;

pq.push(rootNode); //在 priority\_queue 中插入根节点

while (!pq.empty()) {

HeapNode temp; //temp指向堆顶元素

temp = pq.top();

pq.pop();

int level = temp.level;

if ((level == C.size())) {

for (int i = 0; i <= C.size(); i++) { //输出该结点所对应的路径

std::cout << temp.route[i] << " ";

}

std::cout << std::endl << temp.v;

return;

} else {

if ((level == C.size() - 1) && C[temp.route[level] - 1][temp.route[0] - 1] != BIGNUM) {

temp.route.push\_back(temp.route[0]);

float lb = temp.v + C[temp.route[level] - 1][temp.route[0] - 1];

if (lb < upbound) {

HeapNode node;

node.v = lb;

node.bound = lb;

for (int i = 0; i <= level + 1; i++)

node.route.push\_back(temp.route[i]);

node.level = level + 1;

pq.push(node);

upbound = lb;

}

} else {

//在若干个城市中选择

for (int j = 1; j <= C.size(); j++) {

if ((C[temp.route[level] - 1][j - 1] != BIGNUM) && !Contain(temp, j)) {

temp.route.push\_back(j);

temp.level = level + 1;

float lb = ((temp.v + C[temp.route[level] - 1][j - 1]) \* 2 + SiSum(C, temp) + SjSum(C, temp)) / 2.0;

if (lb <= upbound) {

HeapNode node;

node.v = temp.v + C[temp.route[level] - 1][j - 1];

node.bound = lb;

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

node.route.push\_back(temp.route[i]);

node.route.push\_back(j);

node.level = level + 1;

pq.push(node);

}

}

}

}

}

}

}

int main() {

int n;

std::cout << "Enter the number of cities: ";

std::cin >> n;

std::vector<std::vector<int>> C(n, std::vector<int>(n));

int upbound = 0;

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

C[i][i] = BIGNUM;

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

std::cout << "Enter the distance between city " << i+1 << " and city " << j+1 << ": ";

std::cin >> C[i][j];

C[j][i] = C[i][j];

}

}

int k = n - 1;

int i = 0;

int max = BIGNUM;

int row;

std::vector<int> visited(n, 0);

while (k) {

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

if (j == i || visited[j])

continue;

if (C[i][j] < max) {

max = C[i][j];

row = j;

}

}

upbound += max;

visited[i] = 1;

i = row;

k--;

max = BIGNUM;

}

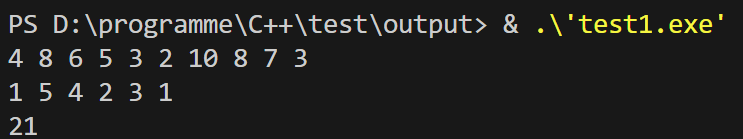
upbound += C[row][0];

Tsp(C, upbound);

return 0;

}

**4.2上机实验**

**4.2.1 运行结果**

**4.2.** **调试程序，给出按照（2）修改后的程序在样例输入情况下执行过程中主要变量的变化情况，并与（1）修改前的主要变量变化情况进行对比，验证修改后的程序是否正确**

4 8 6 5 3 2 10 8 7 3

4 8 6 5 3 2 10 8 7 3

v,level,bound = 0,0,14

v,level,bound = 5,1,18

v,level,bound = 8,2,18

v,level,bound = 10,3,18

v,level,bound = 13,4,18

v,level,bound = 21,5,21

1 5 4 2 3 1

21

**5. 实验总结**

在这个问题中，我们实现了一个动态规划算法来解决旅行商问题（TSP）。我们通过动态分配数组C的大小来允许输入任意多个城市，并且根据城市数量来动态分配数组的大小。然后，我们使用动态规划算法来求解最短回路。

在算法实现中，我们使用了优先队列来存储待处理的节点，并且使用了启发式函数来估计每个节点的上界，以便进行剪枝。我们还实现了一些辅助函数来计算路径长度和边界值，以及一些辅助数据结构来存储路径和已访问的城市。

通过这个实验，我们学习了如何使用动态规划算法解决TSP问题，并且了解了如何根据输入动态分配数组的大小。同时，我们也学习了如何使用优先队列和启发式函数来优化搜索过程，以提高算法的效率。

总的来说，这个实验帮助我们加深了对动态规划算法和TSP问题的理解，同时也提高了我们对动态分配数组大小和优先队列的应用能力。