**2021-11-28 算法作业**

21215122 何峙 大数据与人工智能

1. **局部搜索算法的具体应用：**

以下以TSP旅行商问题为例：

#include <iostream>

#include <cmath>

#include <stdlib.h>

#include <time.h>

#include <vector>

#include <windows.h>

#include <memory.h>

#include <string.h>

#include <iomanip>

#define DEBUG

using namespace std;

#define CITY\_SIZE 52 //城市数量

//城市坐标

typedef struct candidate

{

int x;

int y;

}city, CITIES;

//优化值

int \*\*Delta;

//解决方案

typedef struct Solution

{

int permutation[CITY\_SIZE]; //城市排列

int cost; //该排列对应的总路线长度

}SOLUTION;

// 计算邻域操作优化值

int calc\_delta(int i, int k, int \*tmp, CITIES \* cities);

//计算两个城市间距离

int distance\_2city(city c1, city c2);

//根据产生的城市序列，计算旅游总距离

int cost\_total(int \* cities\_permutation, CITIES \* cities);

//获取随机城市排列, 用于产生初始解

void random\_permutation(int \* cities\_permutation);

//颠倒数组中下标begin到end的元素位置, 用于two\_opt邻域动作

void swap\_element(int \*p, int begin, int end);

//邻域动作 反转index\_i <-> index\_j 间的元素

void two\_opt\_swap(int \*cities\_permutation, int \*new\_cities\_permutation, int index\_i, int index\_j);

//本地局部搜索，边界条件 max\_no\_improve

void local\_search(SOLUTION & best, CITIES \* cities, int max\_no\_improve);

//将城市序列分成4块，然后按块重新打乱顺序。

//用于扰动函数

void double\_bridge\_move(int \*cities\_permutation, int \* new\_cities\_permutation);

//扰动

void perturbation(CITIES \* cities, SOLUTION &best\_solution, SOLUTION &current\_solution);

//迭代搜索

void iterated\_local\_search(SOLUTION & best, CITIES \* cities, int max\_iterations, int max\_no\_improve);

// 更新Delta

void Update(int i, int k, int \*tmp, CITIES \* cities);

//城市排列

int permutation[CITY\_SIZE];

//城市坐标数组

CITIES cities[CITY\_SIZE];

//berlin52城市坐标，最优解7542好像

CITIES berlin52[CITY\_SIZE] = { { 565,575 },{ 25,185 },{ 345,750 },{ 945,685 },{ 845,655 },

{ 880,660 },{ 25,230 },{ 525,1000 },{ 580,1175 },{ 650,1130 },{ 1605,620 },

{ 1220,580 },{ 1465,200 },{ 1530,5 },{ 845,680 },{ 725,370 },{ 145,665 },

{ 415,635 },{ 510,875 },{ 560,365 },{ 300,465 },{ 520,585 },{ 480,415 },

{ 835,625 },{ 975,580 },{ 1215,245 },{ 1320,315 },{ 1250,400 },{ 660,180 },

{ 410,250 },{ 420,555 },{ 575,665 },{ 1150,1160 },{ 700,580 },{ 685,595 },

{ 685,610 },{ 770,610 },{ 795,645 },{ 720,635 },{ 760,650 },{ 475,960 },

{ 95,260 },{ 875,920 },{ 700,500 },{ 555,815 },{ 830,485 },{ 1170,65 },

{ 830,610 },{ 605,625 },{ 595,360 },{ 1340,725 },{ 1740,245 } };

int main()

{

srand(1);

int max\_iterations = 600;

int max\_no\_improve = 50;

//初始化指针数组

Delta = new int\*[CITY\_SIZE];

for (int i = 0; i < CITY\_SIZE; i ++)

Delta[i] = new int[CITY\_SIZE];

SOLUTION best\_solution;

iterated\_local\_search(best\_solution, berlin52, max\_iterations, max\_no\_improve);

cout << endl<<endl<<"搜索完成！ 最优路线总长度 = " << best\_solution.cost << endl;

cout << "最优访问城市序列如下：" << endl;

for (int i = 0; i < CITY\_SIZE;i++)

{

cout << setw(4) << setiosflags(ios::left) << best\_solution.permutation[i];

}

cout << endl << endl;

return 0;

}

//计算两个城市间距离

int distance\_2city(city c1, city c2)

{

int distance = 0;

distance = sqrt((double)((c1.x - c2.x)\*(c1.x - c2.x) + (c1.y - c2.y)\*(c1.y - c2.y)));

return distance;

}

//根据产生的城市序列，计算旅游总距离

//所谓城市序列，就是城市先后访问的顺序，比如可以先访问ABC，也可以先访问BAC等等

//访问顺序不同，那么总路线长度也是不同的

//p\_perm 城市序列参数

int cost\_total(int \* cities\_permutation, CITIES \* cities)

{

int total\_distance = 0;

int c1, c2;

//逛一圈，看看最后的总距离是多少

for (int i = 0; i < CITY\_SIZE; i++)

{

c1 = cities\_permutation[i];

if (i == CITY\_SIZE - 1) //最后一个城市和第一个城市计算距离

{

c2 = cities\_permutation[0];

}

else

{

c2 = cities\_permutation[i + 1];

}

total\_distance += distance\_2city(cities[c1], cities[c2]);

}

return total\_distance;

}

//获取随机城市排列

void random\_permutation(int \* cities\_permutation)

{

int i, r, temp;

for (i = 0; i < CITY\_SIZE; i++)

{

cities\_permutation[i] = i; //初始化城市排列，初始按顺序排

}

for (i = 0; i < CITY\_SIZE; i++)

{

//城市排列顺序随机打乱

r = rand() % (CITY\_SIZE - i) + i;

temp = cities\_permutation[i];

cities\_permutation[i] = cities\_permutation[r];

cities\_permutation[r] = temp;

}

}

//颠倒数组中下标begin到end的元素位置

void swap\_element(int \*p, int begin, int end)

{

int temp;

while (begin < end)

{

temp = p[begin];

p[begin] = p[end];

p[end] = temp;

begin++;

end--;

}

}

//邻域动作 反转index\_i <-> index\_j 间的元素

void two\_opt\_swap(int \*cities\_permutation, int \*new\_cities\_permutation, int index\_i, int index\_j)

{

for (int i = 0; i < CITY\_SIZE; i++)

{

new\_cities\_permutation[i] = cities\_permutation[i];

}

swap\_element(new\_cities\_permutation, index\_i, index\_j);

}

int calc\_delta(int i, int k, int \*tmp, CITIES \* cities){

int delta = 0;

if (i == 0)

{

if (k == CITY\_SIZE - 1)

{

delta = 0;

}

else

{

delta = 0

- distance\_2city(cities[tmp[k]], cities[tmp[k + 1]])

+ distance\_2city(cities[tmp[i]], cities[tmp[k + 1]])

- distance\_2city(cities[tmp[CITY\_SIZE - 1]], cities[tmp[i]])

+ distance\_2city(cities[tmp[CITY\_SIZE - 1]], cities[tmp[k]]);

}

}

else

{

if (k == CITY\_SIZE - 1)

{

delta = 0

- distance\_2city(cities[tmp[i - 1]], cities[tmp[i]])

+ distance\_2city(cities[tmp[i - 1]], cities[tmp[k]])

- distance\_2city(cities[tmp[0]], cities[tmp[k]])

+ distance\_2city(cities[tmp[i]], cities[tmp[0]]);

}

else

{

delta = 0

- distance\_2city(cities[tmp[i - 1]], cities[tmp[i]])

+ distance\_2city(cities[tmp[i - 1]], cities[tmp[k]])

- distance\_2city(cities[tmp[k]], cities[tmp[k + 1]])

+ distance\_2city(cities[tmp[i]], cities[tmp[k + 1]]);

}

}

return delta;

}

// 去重处理

void Update(int i, int k, int \*tmp, CITIES \* cities){

if (i && k != CITY\_SIZE - 1){

i --; k ++;

for (int j = i; j <= k; j ++){

for (int l = j + 1; l < CITY\_SIZE; l ++){

Delta[j][l] = calc\_delta(j, l, tmp, cities);

}

}

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

for (int l = i; l <= k; l ++){

if (j >= l) continue;

Delta[j][l] = calc\_delta(j, l, tmp, cities);

}

}

}// 如果不是边界，更新(i-1, k + 1)之间的

else{

for (i = 0; i < CITY\_SIZE - 1; i++)

{

for (k = i + 1; k < CITY\_SIZE; k++)

{

Delta[i][k] = calc\_delta(i, k, tmp, cities);

}

}

}// 边界要特殊更新

}

//本地局部搜索，边界条件 max\_no\_improve

//best\_solution最优解

//current\_solution当前解

void local\_search(SOLUTION & best\_solution, CITIES \* cities, int max\_no\_improve)

{

int count = 0;

int i, k;

int inital\_cost = best\_solution.cost; //初始花费

int now\_cost = 0;

SOLUTION \*current\_solution = new SOLUTION; //为了防止爆栈……直接new了，你懂的

for (i = 0; i < CITY\_SIZE - 1; i++)

{

for (k = i + 1; k < CITY\_SIZE; k++)

{

Delta[i][k] = calc\_delta(i, k, best\_solution.permutation, cities);

}

}

do

{

//枚举排列

for (i = 0; i < CITY\_SIZE - 1; i++)

{

for (k = i + 1; k < CITY\_SIZE; k++)

{

//邻域动作

two\_opt\_swap(best\_solution.permutation, current\_solution->permutation, i, k);

now\_cost = inital\_cost + Delta[i][k];

current\_solution->cost = now\_cost;

if (current\_solution->cost < best\_solution.cost)

{

count = 0; //better cost found, so reset

for (int j = 0; j < CITY\_SIZE; j++)

{

best\_solution.permutation[j] = current\_solution->permutation[j];

}

best\_solution.cost = current\_solution->cost;

inital\_cost = best\_solution.cost;

Update(i, k, best\_solution.permutation, cities);

}

}

}

count++;

} while (count <= max\_no\_improve);

}

//将城市序列分成4块，然后按块重新打乱顺序。

//用于扰动函数

void double\_bridge\_move(int \*cities\_permutation, int \* new\_cities\_permutation)

{

int temp\_perm[CITY\_SIZE];

int pos1 = 1 + rand() % (CITY\_SIZE / 4);

int pos2 = pos1 + 1 + rand() % (CITY\_SIZE / 4);

int pos3 = pos2 + 1 + rand() % (CITY\_SIZE / 4);

int i;

vector<int> v;

//第一块

for (i = 0; i < pos1; i++)

{

v.push\_back(cities\_permutation[i]);

}

//第二块

for (i = pos3; i < CITY\_SIZE; i++)

{

v.push\_back(cities\_permutation[i]);

}

//第三块

for (i = pos2; i < pos3; i++)

{

v.push\_back(cities\_permutation[i]);

}

//第四块

for (i = pos1; i < pos2; i++)

{

v.push\_back(cities\_permutation[i]);

}

for (i = 0; i < (int)v.size(); i++)

{

new\_cities\_permutation[i] = v[i];

}

}

//扰动

void perturbation(CITIES \* cities, SOLUTION &best\_solution, SOLUTION &current\_solution)

{

double\_bridge\_move(best\_solution.permutation, current\_solution.permutation);

current\_solution.cost = cost\_total(current\_solution.permutation, cities);

}

//迭代搜索

//max\_iterations用于迭代搜索次数

//max\_no\_improve用于局部搜索边界条件

void iterated\_local\_search(SOLUTION & best\_solution, CITIES \* cities, int max\_iterations, int max\_no\_improve)

{

SOLUTION \*current\_solution = new SOLUTION;

//获得初始随机解

random\_permutation(best\_solution.permutation);

best\_solution.cost = cost\_total(best\_solution.permutation, cities);

local\_search(best\_solution, cities, max\_no\_improve); //初始搜索

for (int i = 0; i < max\_iterations; i++)

{

perturbation(cities, best\_solution, \*current\_solution); //扰动+判断是否接受新解

local\_search(\*current\_solution, cities, max\_no\_improve);//继续局部搜索

//找到更优解

if (current\_solution->cost < best\_solution.cost)

{

for (int j = 0; j < CITY\_SIZE; j++)

{

best\_solution.permutation[j] = current\_solution->permutation[j];

}

best\_solution.cost = current\_solution->cost;

}

cout << setw(13) << setiosflags(ios::left) <<"迭代搜索 " << i << " 次\t" << "最优解 = " << best\_solution.cost << " 当前解 = " << current\_solution->cost << endl;

}

}

1. **实现局部搜索的优化：选择概率算法**

用以上旅行商问题，作如下改动：

#include <algorithm>

#include <cmath>

#include <cstdio>

#include <cstring>

#include <deque>

#include <iostream>

#include <map>

#include <queue>

#include <set>

#include <stack>

#include <string>

#include <vector>

using namespace std;

typedef long long LL;

const int maxn = 1e2 + 7;

const int INF = 0x7fffffff;

const double PI = acos(-1);

struct Point { //点类

string name;

double x, y;

int i; //编号

};

vector<Point> p;

double d[maxn][maxn]; //距离矩阵

int n;

double sum = 0; //当前最短路径长度

double dist(Point a, Point b) { //计算两点距离

return sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

double get\_sum(vector<Point> a) { //返回路径长度

double sum = 0;

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

sum += d[a[i].i][a[i - 1].i];

}

sum += d[a[0].i][a[a.size()-1].i];

return sum;

}

void init() { //初始化

srand((unsigned)time(NULL)); //设置随机数种子

cin >> n;

p.clear();

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

Point t;

cin >> t.name >> t.x >> t.y;

t.i = i;

p.push\_back(t);

}

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

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

d[i][j] = d[j][i] = dist(p[i], p[j]);

}

}

sum = get\_sum(p);

}

void show() { //显示当前结果

cout << "路径长度: " << sum << endl;

cout << "路径:";

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

cout << ' ' << p[i].name;

puts("");

}

int w = 100;

vector<vector<Point>> group;

void Improve\_Circle() { //改良圈法得到初始序列

vector<Point> cur = p;

for (int t = 0; t < w; t++) { //重复50次

for (int i = 0; i < n; i++) { //构造随机顺序

int j = rand() % n;

swap(cur[i], cur[j]);

}

int flag = 1;

while (flag) {

flag = 0;

//不断选取uv子串，尝试倒置uv子串的顺序后解是否更优，如果更优则变更

for (int u = 1; u < n - 2; u++) {

for (int v = u + 1; v < n - 1; v++) {

if (d[cur[u].i][cur[v + 1].i] + d[cur[u - 1].i][cur[v].i] <

d[cur[u].i][cur[u - 1].i] + d[cur[v].i][cur[v + 1].i]) {

for (int k = u; k <= (u + v) / 2; k++) {

swap(cur[k], cur[v - (k - u)]);

flag = 1;

}

}

}

}

}

group.push\_back(cur);

double cur\_sum = get\_sum(cur);

if (cur\_sum < sum) {

sum = cur\_sum;

p = cur;

}

}

}

vector<int> get\_randPerm(int n) { //返回一个随机序列

vector<int> c;

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

c.push\_back(i);

}

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

swap(c[i], c[rand() % n]);

}

return c;

}

//排序时用到的比较函数

bool cmp(vector<Point> a, vector<Point> b) { return get\_sum(a) < get\_sum(b); }

int dai = 200; //一共进行200代的进化选择

int c[maxn];

double bylv = 0.1;

void genetic\_algorithm() { //选择概率算法

vector<vector<Point>> A = group, B, C;

for (int t = 0; t < dai; t++) {

B = A;

vector<int> c = get\_randPerm(A.size());

for (int i = 0; i + 1 < c.size(); i += 2) {

int F = rand() % n;

int u=c[i],v=c[i+1];

for (int j = F; j < n;

j++) {

swap(B[u][j], B[v][j]);

}

int num1[1000]={0},num2[1000]={0};

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

num1[B[u][j].i]++;

num2[B[v][j].i]++;

}

vector<Point> v1;

vector<Point> v2;

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

if(num1[B[u][j].i]==2){

v1.push\_back(B[u][j]);

}

}

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

if(num2[B[v][j].i]==2){

v2.push\_back(B[v][j]);

}

}

int p1=0,p2=0;

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

if(num1[B[u][j].i]==2){

B[u][j]=v2[p2++];

}

if(num2[B[v][j].i]==2){

B[v][j]=v1[p1++];

}

}

}

C.clear();

int flag=1;

for (int i = 0; i < A.size(); i++) {

if (rand() % 100 >= bylv \* 100)

continue;

//对于变异的个体,取3个点u<v<w,把子串[u,v]插到w后面

int u, v, w;

u = rand() % n;

do {

v = rand() % n;

} while (u == v);

do {

w = rand() % n;

} while (w == u || w == v);

if (u > v)

swap(u, v);

if (v > w)

swap(v, w);

if (u > v)

swap(u, v);

vector<Point> vec;

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

vec.push\_back(A[i][j]);

for (int j = v; j < w; j++)

vec.push\_back(A[i][j]);

for (int j = u; j < v; j++)

vec.push\_back(A[i][j]);

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

vec.push\_back(A[i][j]);

C.push\_back(vec);

}

//合并A，B，C

for (int i = 0; i < B.size(); i++) {

A.push\_back(B[i]);

}

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

A.push\_back(C[i]);

}

sort(A.begin(), A.end(), cmp); //从小到大排序

vector<vector<Point>> new\_A;

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

new\_A.push\_back(A[i]);

}

A = new\_A;

}

group = A;

sum = get\_sum(group[0]);

p = group[0];

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("in.txt", "r", stdin);

#endif

init();

cout << "初始";

show();

cout << "改良圈法";

Improve\_Circle();

show();

cout << "选择概率算法";

genetic\_algorithm();

show();

return 0;

}

1. **实现局部搜索的优化：模拟退火算法**

还是用旅行商问题，作如下改动：

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#include<math.h>

#define T0 50000.0 // 初始温度

#define T\_end (1e-8)

#define q 0.98 // 退火系数

#define L 1000 // 每个温度时的迭代次数，即链长

#define N 31 // 城市数量

int city\_list[N]; // 用于存放一个解

// 城市坐标

double city\_pos[N][2] =

{

{1304,2312},{3639,1315},{4177,2244},{3712,1399},

{3488,1535},{3326,1556},{3238,1229},{4196,1004},

{4312,790},{4386,570},{3007,1970},{2562,1756},

{2788,1491},{2381,1676},{1332,695},

{3715,1678},{3918,2179},{4061,2370},

{3780,2212},{3676,2578},{4029,2838},

{4263,2931},{3429,1908},{3507,2367},

{3394,2643},{3439,3201},{2935,3240},

{3140,3550},{2545,2357},{2778,2826},

{2370,2975}};

//函数声明

double distance(double \*,double \*); // 计算两个城市距离

double path\_len(int \*); // 计算路径长度

void init(); //初始化函数

void create\_new(); // 产生新解

// 距离函数

double distance(double \* city1,double \* city2)

{

double x1 = \*city1;

double y1 = \*(city1+1);

double x2 = \*(city2);

double y2 = \*(city2+1);

double dis = sqrt((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));

return dis;

}

// 计算路径长度

double path\_len(int \* arr)

{

double path = 0; // 初始化路径长度

int index = \*arr; // 定位到第一个数字(城市序号)

for(int i=0;i<N-1;i++)

{

int index1 = \*(arr+i);

int index2 = \*(arr+i+1);

double dis = distance(city\_pos[index1-1],

city\_pos[index2-1]);

path += dis;

}

int last\_index = \*(arr+N-1); // 最后一个城市序号

int first\_index = \*arr; // 第一个城市序号

double last\_dis = distance(city\_pos[last\_index-1],

city\_pos[first\_index-1]);

path = path + last\_dis;

return path; // 返回总的路径长度

}

// 初始化函数

void init()

{

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

city\_list[i] = i+1; // 初始化一个解

}

// 产生一个新解

// 此处采用随机交换两个位置的方式产生新的解

void create\_new()

{

double r1 = ((double)rand())/(RAND\_MAX+1.0);

double r2 = ((double)rand())/(RAND\_MAX+1.0);

int pos1 = (int)(N\*r1); //第一个交叉点的位置

int pos2 = (int)(N\*r2);

int temp = city\_list[pos1];

city\_list[pos1] = city\_list[pos2];

city\_list[pos2] = temp; // 交换两个点

}

// 主函数

int main(void)

{

srand((unsigned)time(NULL)); //初始化随机数种子

time\_t start,finish;

start = clock(); // 程序运行开始计时

double T;

int count = 0; // 记录降温次数

T = T0; //初始温度

init(); //初始化一个解

int city\_list\_copy[N]; // 用于保存原始解

double f1,f2,df; //f1为初始解目标函数值，

//f2为新解目标函数值，df为二者差值

double r; // 0-1之间的随机数，用来决定是否接受新解

while(T > T\_end) // 当温度低于结束温度时，退火结束

{

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

{

// 复制数组

memcpy(city\_list\_copy,city\_list,N\*sizeof(int));

create\_new(); // 产生新解

f1 = path\_len(city\_list\_copy);

f2 = path\_len(city\_list);

df = f2 - f1;

// 以下是Metropolis准则

if(df >= 0)

{

r = ((double)rand())/(RAND\_MAX);

if(exp(-df/T) <= r) // 保留原来的解

{

memcpy(city\_list,city\_list\_copy,N\*sizeof(int));

}

}

}

T \*= q; // 降温

count++;

}

finish = clock(); // 退火过程结束

double duration = ((double)(finish-start))/CLOCKS\_PER\_SEC; // 计算时间

printf("模拟退火算法，初始温度T0=%.2f,降温系数q=%.2f,每个温度迭代%d次,共降温%d次，得到的TSP最优路径为:\n",T0,q,L,count);

for(int i=0;i<N-1;i++) // 输出最优路径

{

printf("%d--->",city\_list[i]);

}

printf("%d\n",city\_list[N-1]);

double len = path\_len(city\_list); // 最优路径长度

printf("最优路径长度为:%lf\n",len);

printf("程序运行耗时:%lf秒.\n",duration);

return 0;

}