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;
    }

}
```

2. 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;
}
```

3. 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;
}
```