遗传算法求解 TSP 问题

摘要：

TSP 问题中文译名为旅行商问题，该问题是图论中最著名的难题之一 ： 即给出一个具有 n 个顶点的完全图，每条边的权值为边的两个端点之间的直线距离，求总权值最小的经过每个顶点正好一次的封闭回路。该问题已被证明具有 NPC 计算复杂性，对于这类问题的大型实例不能用精确算法求解,必须寻求这类问题的有效的近似算法。遗传算法就是一种可以有效求得该问题近似解的算法。

关键词： 遗传算法 TSP 智能算法 智能搜索

一. 遗传算法概述

在看过的几本介绍遗传算法的书籍上，开篇都是先大段大段的讲述自然界中生物繁殖进化的过程，因为遗传算法的实质就是在模拟生物从繁殖，到变异，然后经过进化和自然选择的全过程。遗传算法有别于传统的确定性算法之处在于，它并没有给出解决某一问题的确定方法，它只是用 ”基因组” 来表示问题的结果， 然后在这些基因中选择出一部分最优(最靠近真实解)的基因组，然后让这些基因组相互杂交来产生后代(新基因组)，这些新产生的后代还有一定的机率产生变异，从而可以进一步产生新的基因组。遗传算法中非常重要的一步是进行选择，即从当前种群中按照一定的方法选择一部分个体进行杂交繁殖后代。 选择条件是否恰当合理，直接决定了基因进化的方向，因此对结果的影响非常大。选择操作没做好，可能导致最后得到的解只是局部最优解，离全局最优解偏差较大。另外一个需要注意的问题是，在进行杂交和变异时，产生的新基因组是否是合法的， 也就是说，产生的新基因组能否表示一种有效合法的可行解。如果不能的话，就要考虑寻找其他的杂交或者变异的方法。最后还有一个要注意的点是，如何能确定遗传算法已经找到了最优解，不过不能确定是否找到了最优解，那么该何时终止算法。

二. 将遗传算法应用于 TSP 问题

可以采用搜索找到答案的问题，基本上都可以采用遗传算法解决。当然有些不能采用搜索解决的问题也可以使用遗传算法。小规模的 TSP 问题可以采用全排列算法配合剪枝等优化手段可以求得最优解，但是全排列算法的时间复杂度是 O(N!)，这是时间复杂度随问题规模增长得最快的一种量级。随着数据规模的增加，时间复杂度急剧增长，当 N 为 13 左右时，就能明显看到算法求解时间变慢了许多，当 N 继续增长时，求解时间将不可接受。而本次实践中采用的遗传算法，求解 N 为 50 的问题规模时，依然能很快计算出问题的一个解。

将遗传算法应用于某一个问题，一般要经过以下几个步骤：

1. 确定如何用基因组表示问题的解
2. 确定采用何种数据结构来保存基因组
3. 如何衡量和计算基因组的适应度
4. 采用何种方法进行“自然选择”
5. 如何对选择出来的基因组进行杂交，使得杂交出来的基因组是有效合法的。
6. 怎么使基因组进行“基因突变”，使得突变后的基因组依然是合法有效的。

下面就将依照这六步，来讲解如何用遗传算法求解 TSP 问题。

1.确定如何用基因组表示问题的解

这需要具体问题具体分析，在 TSP 问题中，如果我们将所有顶点(表示原问题中的城市)从 0 到 N-1 进行编号，由于问题需要找到一条正好经过每一个顶点一次的最短封闭回路，那么问题的解实际上可以表示为 0 到 N-1 的一个排列， 不同的排列表示不同的路径，这样一来，我们就可以用 0 到 N-1 这 N 个数字的一个排列表示一个基因组，一个基因组表示一条可行的路径。

2.确定采用何种数据结构来保存基因组

在上一步中，我们确定了基因组是 N 个确定数字的某种排列，这在逻辑上是一种线性结构，所以我们需要使用一种线性数据结构来保存基因组。线性数据结构有数组，链表和顺序表，为了操作方便，我们采用顺序表 vector 来保存基因组。下面是本项目中编写的基因组结构体。需要提一下的是，在遗传算法的开始，需要产生一群具有不同基因组的个体，因此在构造函数中用随机函数打乱了基因的顺序以使得初始种群具有一定的多样性。

```
struct Genome{
    vector<int> mv_gene; //use vector to store the gene
    double md_fitness; //the fitness of this genome
    Genome() {}
    Genome(int n) {
        mv_gene.reserve(n);
        for (int i = 0; i < n; ++i) mv_gene.push_back(i);
        for (int i = 0; i < n; ++i) {
            int x = randInt(0, n - 1);
            int y = randInt(0, n - 1);
            if (x == y) continue;
            swap(mv_gene[x], mv_gene[y]);
        }
    }

    int at(int k) const { return mv_gene[k]; }
    int& operator[](int n) { return mv_gene[n]; }
    int size() { return mv_gene.size(); }
    bool operator<(const Genome& g) const{ return md_fitness < g.md_fitness; }
    bool operator==(const Genome& g)const { return mv_gene == g.mv_gene; }
};
```

3.如何衡量和计算基因的适应度

这个也是需要具体问题具体分析的，在这个 TSP 问题中，我们需要求得的是**一条经过每个城市一次且长度最短的回路。**需要重点关注的是长度最短，所以我们应该用每条道路的长度信息计算每个基因组的适应度。具体方法出：先计算出所以基因所代表的路径的长度，然后每个基因的适应度定义为最长路径的长度与当前路径的长度的差值。很显然，这样得出的适应度，回路长度最短的基因将具有最高的适应度，而回路长度最长的基因的适应度将为 0。对于这个问题而言，这是一个合理的计算适应度的方法。程序中的代码如下：

```
void calculatePopulationFitness() {
    double longest = 0;
    for (int i = 0, n = mv_popu.size(); i < n; ++i) {
        mv_popu[i].md_fitness = mp_map->lengthOfRouteSquare(mv_popu[i].mv_gene);
        if (mv_popu[i].md_fitness > longest)  longest = mv_popu[i].md_fitness;
        md_totalFitness += mv_popu[i].md_fitness;
    }
    md_aveFitness = md_totalFitness / mv_popu.size();

    //calculate fitness
    for (int i = 0, n = mv_popu.size(); i < n; ++i) {
        mv_popu[i].md_fitness = longest - mv_popu[i].md_fitness;
        if (mv_popu[i].md_fitness > md_bestFitness) md_bestFitness = mv_popu[i].md_fitness;
    }

    //if more then 80% of population have the same value(length of route)
    //with the best one,then stop
    //if ((md_totalFitness / (longest*mv_popu.size())) >= 0.80) mb_running = false;

}
```

其中的 lengthOfRouteSquare()函数接受一个基因组作为参数，然后计算并返回这个基因组中保存的城市序列所对应的回路的长度，为了节省 CPU 时间， 这里没有对距离进行开方运算。

4.采用何种方法进行“自然选择”

在物质繁殖的过程中，“自然选择”对物种的进化起到了非常重要的作用。它使得适应环境的个体以**很大几率**存活下来，而那些不太适应环境的个体**很可能**会被淘汰。注意我这里的用词，“很大几率”和“很可能”，而不是“一定会”。打个比方说：假如某种青蛙很耐热，如果在夏天的时候那些不太耐热的青蛙全部都被热死了，剩下的都是非常耐热的青蛙(局部最优解)，那么到了冬天，这些非常耐热的青蛙也很有可能全部都被冻死了。这样一来，这种青蛙不是很容易就会灭绝了吗。所以在选择的时候不能太过与绝对，这样就容易使得程序得到局部最优解而非全局最优解。在本项目中， 采用了三种选择方法，程序在运行的时候将会随机地采用这三种中的一种进行选择。

a. 赌轮盘选择法

相信所有人都看过旋转轮盘进行抽奖的画面，玩家将轮盘用力往某个方向旋转一下，轮盘停下来后，指针所指的位置就是抽奖的结果。在实际场景中，为了公平起见，轮盘上的每一格大小都是相等的。但是在这里，我们需要改变一下游戏规则，因为我们需要让适应度跟大的个体以更大的几率被选中进行繁殖后代，因为他们具有更可能得到问题的解的基因。为了达到这样的效果，我们将每个个体(基因)的适应度在所有个体总的适应度中所占的比例做为该个体在轮盘中的夹角所占圆盘的比例。具体代码如下：

```
Genome selectRoulette(vector<Genome>& src) {
    double fSlice = randFloat() * md_totalFitness;
    double sum = 0;
    for (int i = 0, n = src.size(); i < n; ++i) {
        sum += src[i].md_fitness;
        if (sum >= fSlice) return src[i];
    }
}
```

其中 md_totalFitness 是已经计算好的种群的适应度总和。

b. 锦标赛选择法一

锦标赛选择法是在种群中随机抽取 n 个个体，最后选择这 n 个中适应度最高的那一个。

```
//select the best one in this n trys
Genome selectTournament(int numToTry, vector<Genome>& src) {
    int best = randInt(0, src.size()-1);
    for (int i = 0; i < numToTry; ++i) {
        int thisTry = randInt(0, src.size() - 1);
        if (src[best].md_fitness < src[thisTry].md_fitness) best = thisTry;
    }
    return src[best];
}
```

这种选择方法实现起来非常简单，但其中的参数 n 是这种选择方法的关键，实践中常需要多次修改和尝试才能找到一个合适的值。

c. 锦标赛选择法二

这种选择方法与上一种选择方法非常相似，不同之处在于，这种选择方法每次只随机抽取两个个体，让后以一定的概率选择适应度较高或者较低的那个，例如：随机产生一个[0,1)之间的随机值，当该随机值小于 0.8 时，选择适应度较高的那个，当该随机值大于等于 0.8 时，选择适应度较小的那个。具体代码如下：

```
//select the better one of the randomly selected two Genome in 80% percentage,
//and the weaker one in 20% percentage.
Genome selectALTTournament(vector<Genome>& src) {
    int g1 = randInt(0, src.size() - 1);
    int g2 = randInt(0, src.size() - 1);
    while (g1 == g2) g2 = randInt(0, src.size() - 1);
    double r = randFloat();
    if (r < 0.80) {
        return src[g1] < src[g2] ? src[g2] : src[g1];
    }
    else {
        return src[g1] < src[g2] ? src[g1] : src[g2];
    }
}
```

该选择方法也非常简单，计算起来速度也很快。这种选择方法的关键在于找到一个合适的值作为上述描述中的概率值，实践中，也是需要通过多此修改和测试才能找到一个恰当的值，不过一般都落在 0.7 到 0.8 之间。

5.如何对选择出来的基因组进行杂交，使得杂交出来的基因组是有效合法的。

杂交操作就是对选择出来的两个父代基因组，将其基因进行混合，混合后产生的新基因组做为子代的基因，放在染色体中作为子代的遗传物质。需要注意的是，混合后产生的新基因，必须要是合法的，它必须能表示问题一种可行的解。否则这样的杂交操作将不可取。本项目里面实现了种三种杂交操作:

a.部分映射杂交

这种杂交操作的具体过程为，在父代基因组中随机任意选一段基因，对该段基因中来自父亲的基因 GFi 和来自母亲的基因 GMi，他们均在各自软色体的第 i 个位置。现在将他们看成一对，然后，将父亲中的这对基因互换位置。同样，将母亲中的这对基因也互换位置。然后对刚才任意选中的这一段基因中的其他每一对基因，都做相同的操作，即完了此次杂交操作。此时，GFi 和 GMi 各自产生了一个新的基因组合。代码实现如下：

```
*randomly choose a range, for each pair(one from mama and one from dada) of gene
* in the range, swap their position in genome for both baby
*/
void crossoverPMX(Genome& dad, Genome& mum, Genome &baby1, Genome &baby2) {
    baby1 = dad;
    baby2 = mum;
    if (dad == mum) return;
    int beg = randInt(0, dad.size()-1);
    int end = randInt(beg, dad.size() - 1);
    for (int pos = beg; pos <= end; ++pos) {
        int gene1 = dad[pos];
        int gene2 = mum[pos];
        if (gene1 == gene2) continue;
        int pos1, pos2;
        //find and swap the position of gene1 and gene2
        for (int i = 0, n = baby1.size(); i < n; ++i) {
            if (baby1[i] == gene1) pos1 = i;
            else if (baby1[i] == gene2) pos2 = i;
        }
        swap(baby1[pos1], baby1[pos2]);
        //and for baby2
        for (int i = 0, n = baby2.size(); i < n; ++i) {
            if (baby2[i] == gene1) pos1 = i;
            else if (baby2[i] == gene2) pos2 = i;
        }
        swap(baby2[pos1], baby2[pos2]);
    }
}
```

b.基于顺序的杂交

基于顺序的杂交的操作过程为，在父代的基因组中，随机选择一些位置上的单个的基因，然后将父亲中这些位置上的基因的**顺序**强加到母亲的基因组中去，其他位置上的基因保持不变，产生一个子代基因组；同样地，将母亲中相同位置上的基因的**顺序**强加到父亲的基因组中去，其他位置上的基因的也保持不变，又产生了另一个子代基因组。杂交完成。代码实现如下：

```
/*order based crossover
* randomly choose some position in the genome, apply the order of the selected
* gene in dad to mum, and vice versus.
*/
void crossoverOBX(Genome& dad, Genome& mum, Genome &baby1, Genome &baby2) {
    baby1 = dad;
    baby2 = mum;
    if (mum == dad) return;
    vector<int> cities;
    vector<int> positions;
    int n = dad.size();

    int pos = randInt(0, n - 2);
    //randomly choose some cities from dad and record their positions
    while (pos < n) {
        positions.push_back(pos);
        cities.push_back(dad[pos]);
        //notice the range, the return value must >=1,otherwise the same cities
        //maybe choosed twice, which will result in problem
        pos += randInt(1, n - pos);
    }
    //now apply the order of the selected cities to mum to generate baby2
    pos = 0;
    for (int cit = 0; cit < n; ++cit) {
        for (int i = 0; i < cities.size(); ++i) {
            if (baby2[cit] == cities[i]) {
                baby2[cit] = cities[pos++];
                break;
            }
        }
    }

    //now grab the cooresponding cities from mum
    cities.clear();
    for (int i = 0; i < positions.size(); ++i) {
        cities.push_back(mum[positions[i]]);
    }

    //and impose their order in dad to generate baby1
    pos = 0;
    for (int cit = 0; cit < n; ++cit) {
        for (int i = 0; i < cities.size(); ++i) {
            if (baby1[cit] == cities[i]) {
                baby1[cit] = cities[pos++];
                break;
            }
        }
    }
}
```

c.基于位置的杂交

基于位置的杂交的操作过程为，在父代基因组中随机选择一些位置上的单个基因，将这些基因复制到子代中同样的位置上去，剩下的位置用另一个亲代中没有使用的基因按照其原有的顺序逐个填充到该基因组中去，从而产生一个新的基因组。对另外一个亲代也做同样的操作，也产生另外一个新基因组。实现代码如下：

```
/*position based crossover
* randomly select some positions, the cities in those positions will  be
* copied to offsprings to the same positions , the rest positions will be fill
* with cities from the other parent.
*/
void crossoverPBX(Genome& dad, Genome& mum, Genome &baby1, Genome &baby2) {
    if (dad == mum) {
        baby1 = dad;
        baby2 = mum;
        return;
    }
    baby1.mv_gene.reserve(mum.size());
    baby2.mv_gene.reserve(mum.size());
    baby1.mv_gene.assign(mum.size(), -1);
    baby2.mv_gene.assign(mum.size(), -1);

    //randomly select some positions, the cities in those position will  be
    //copied to offsprings to the same position
    vector<int> positions;
    for (int n = mum.size(), pos = randInt(0, n - 2); pos < n; pos += randInt(0, n - pos)) {
        positions.push_back(pos);
    }


      //copy the selected cities to offsprings to the same position
      for (int i = 0, n = positions.size(); i < n; ++i) {
          //baby1 receives selected cities from dad
          baby1[positions[i]] = dad[positions[i]];
          //baby2 receives selected cities from mum
          baby2[positions[i]] = mum[positions[i]];
      }

      //fill the rest positions with cities from the other parent.
      int c1=0, c2=0;
      for (int n = mum.size(), pos = 0; pos < n; ++pos) {
          //baby1's free positions will be filled with cities from mum
          while (c1 < n && baby1[c1] > -1) ++c1; //find a free position
          if (c1<n && find(baby1.mv_gene.begin(), baby1.mv_gene.end(), mum[pos])
              == baby1.mv_gene.end()) {
              baby1[c1] = mum[pos];
          }

          //baby2's free positions will be filled with cities from dad
          while (c2 < n && baby2[c2] > -1) ++c2;
          if (c2<n && find(baby2.mv_gene.begin(), baby2.mv_gene.end(), dad[pos])
              == baby2.mv_gene.end()) {
              baby2[c2] = dad[pos];
          }
      }
  }
```

6.怎么使基因组进行“基因突变”，使得突变后的基因组依然是合法有效的。

和基因杂交一样，基因突变也要考虑突变后产生的新基因组是否也能表示一个合法的解。当突变后产生的新基因组不能表示一个可行的解时，就不能采用这种突变方法。对于 TSP 问题，其基于被设计成为 0 到 n-1 这 n 个数字的某种排列，所以，我们的突变不能产生除 0 到 n-1 之外的数字，也不能增加或者减少数字，只能将当前排列改变为另一种合法的排列。在本项目中，采用了如下四种突变方法：

a. 交换变异

在基因组中随机地选择两个不同位置上的单个基因，然后互换其位置。

```
//randomly choose two genes and swap their position.
void mutateEM(Genome& genome) {
    int gene1 = randInt(0, genome.size() - 1);
    int gene2 = randInt(0, genome.size() - 1);
    while (gene1 == gene2) {
        gene2 = randInt(0, genome.size() - 1);
    }
    swap(genome[gene1], genome[gene2]);
}
```

b. 位移变异

在基因组中随机任意选择一段基因，将其取出来，然后在剩余的基因中随机地选择一个位置将这段基因插进去。代码实现如下：

```
//randomly choose a segment of gene and move it to another random place
void mutateDM(Genome& genome) {
    //make sure that S < E < genome.size()
    int S = randInt(0, genome.size() - 2);
    int E = randInt(S+1, genome.size() - 1);
    vector<int>::iterator beg = genome.mv_gene.begin() + S;
    vector<int>::iterator end = genome.mv_gene.begin() + E;
    vector<int> seg(beg, end);
    genome.mv_gene.erase(beg, end);
    vector<int>::iterator insertPos = genome.mv_gene.begin() + randInt(0, genome.size()-1)
    genome.mv_gene.insert(insertPos, seg.begin(), seg.end());
}
```

c. 插入变异

在基因中随机地选择单个基因，然后将其随机地移动到其他位置上去。这有点像位移变异，所不同的是，位移变异一次移动的是一段基因，插入变异移动的是单个的基因。

```
//randomly choose a gene and move it to another random place.
void mutateIM(Genome& genome) {
    //randomly choose a gene
    vector<int>::iterator pos = genome.mv_gene.begin() + randInt(0, genome.size()-1);
    int city = *pos; //get it
    //remove it from the genome
    genome.mv_gene.erase(pos);
    //randomly choose a place
    pos = genome.mv_gene.begin() + randInt(0, genome.size() - 1);
    //insert it to this place
    genome.mv_gene.insert(pos, city);
}
```

d. 散播变异

在基因组中随机地选择一对位置，将其间的城市进行任意的移动以改变其原有的顺序.代码实现如下：

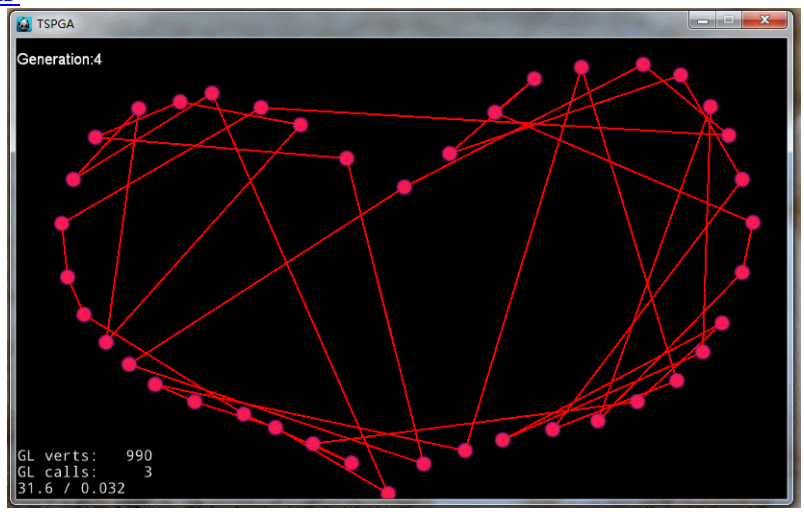
```
//randomly choose a gene and disturbs its order randomly
void mutateSM(Genome& genome) {
    int beg = randInt(0, genome.size()-1);
    int end = randInt(0, genome.size()-1);
    while (beg == end) end = randInt(0, genome.size()-1);
    if (beg > end) swap(beg, end);
    //if there are noly two genes in the segment, just swap this two
    if (end-beg == 1) {
        swap(genome[beg], genome[end]);
        return;
    }
    //swap cnt times, cnt is half length of the range
    for (int i = 0, cnt = (end - beg + 1) / 2; i<cnt; ++i) {
        int x = randInt(beg, end);
        int y = randInt(beg, end);
        if (x == y) continue;
        swap(genome[x], genome[y]);
    }
}
```

三. 在 cocos2dx 游戏引擎中对求解过程可视化

用遗传算法求解某些问题时，往往无法直观地显示出其求解的过程。但庆幸的是，在 TSP 问题中我们能够很好地将其求解过程可视化，也就是说，动态地显示出当前程序的执行过程，以便清楚直观地看到遗传算法的执行进度和采用不同的优化手段时是否能明显提升遗传算法的性能。下面介绍一下到底是怎样对该问题进行可视化的。

我们知道 TSP 问题就是要在一群城市中寻找一条路线长度最短的经过每个城市恰好一次的封闭回路。那么我们就可以在屏幕上创建一些点表示该处有一座城市， 创建完足够的城市之后，我们用城市在屏幕中以像素为单位的坐标位置作为其真实位置，在计算两个城市之间的距离时，计算的就是这两个以像素为单位的顶点的距离。然后在遗传算法的执行过程中，我们记录下种群中适应度最高的个体的基因组。当每一轮执行完成之后，我们从这个最优的基因组中即可得到一个满足题目条件的解，也就是经过每一个城市一次且长度最短的封闭回路，我们把它画出来显示在屏幕上，每一轮结束后，都要更新屏幕上显示的这条回路，当程序运行起来的时候， 就能看到屏幕上的回路在不断变化。这样，我们就完成了遗传算法求解 TSP 问题的可视化。

效果图 n=41.GIF

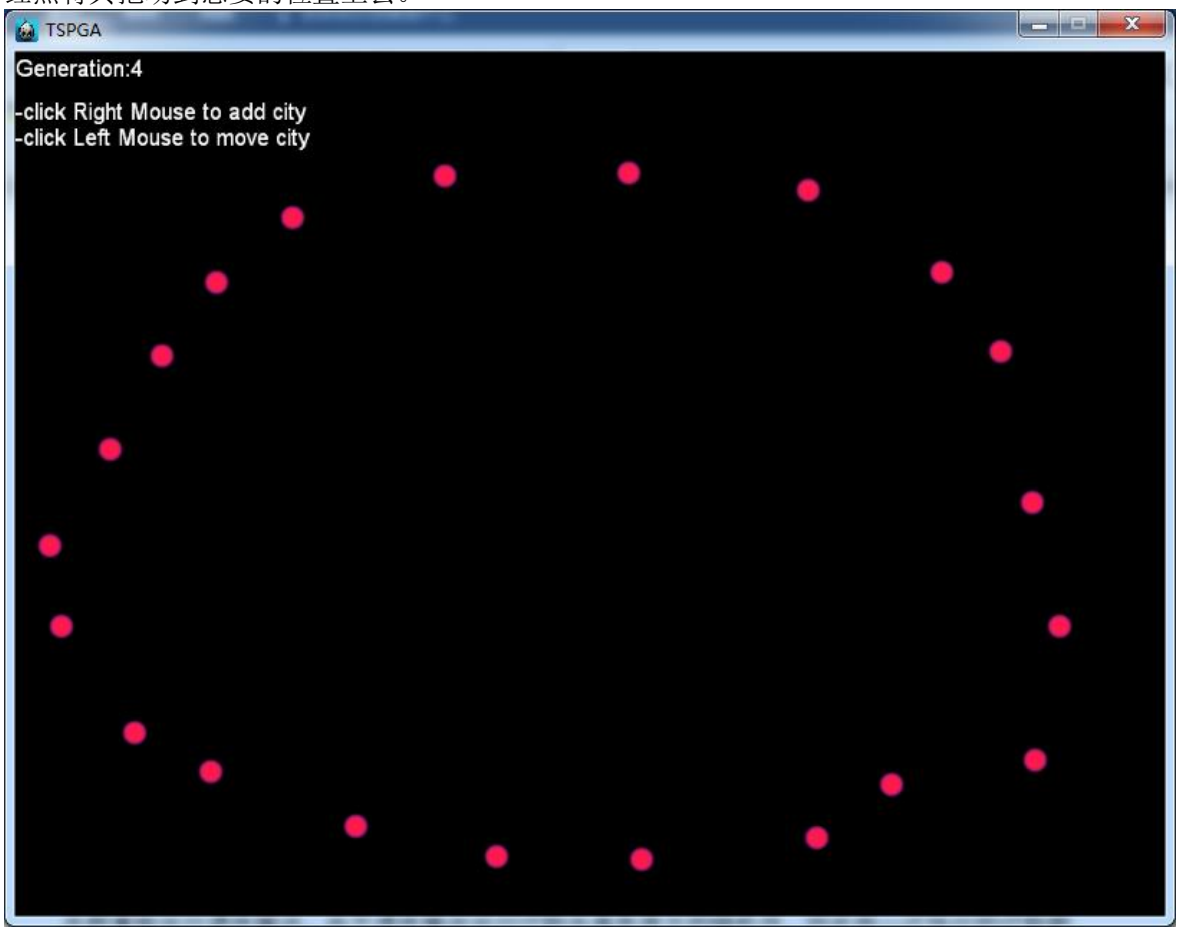


效果图 n=58.GIF

四. 可执行程序使用方法

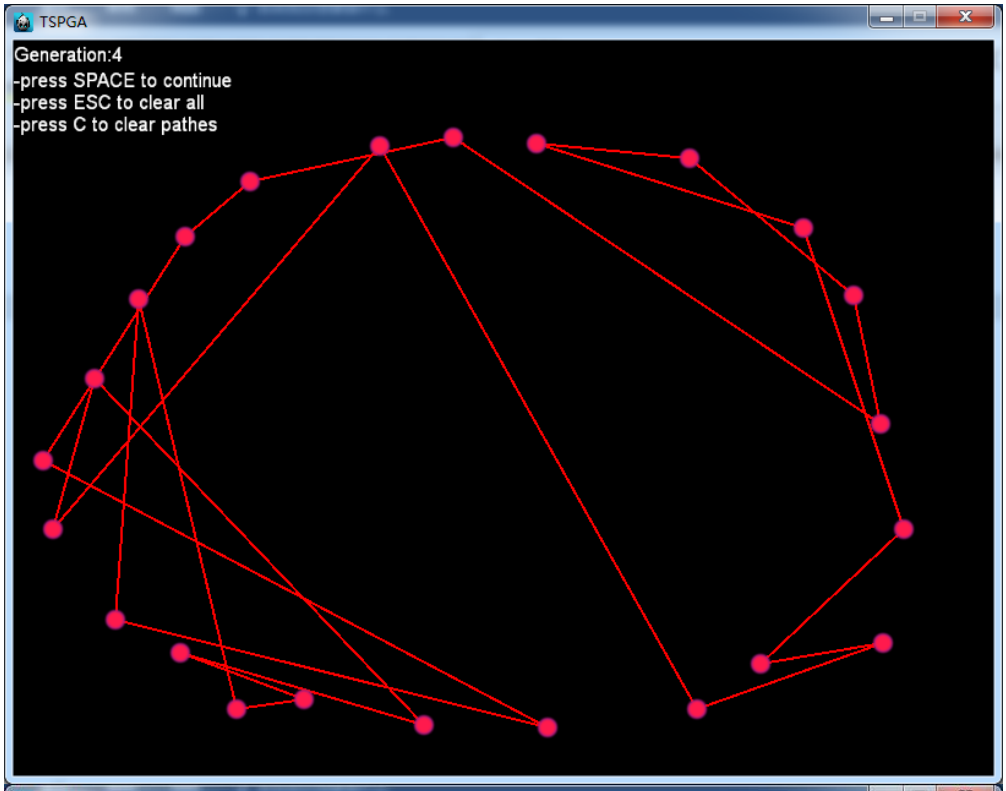
a.创建和移动城市

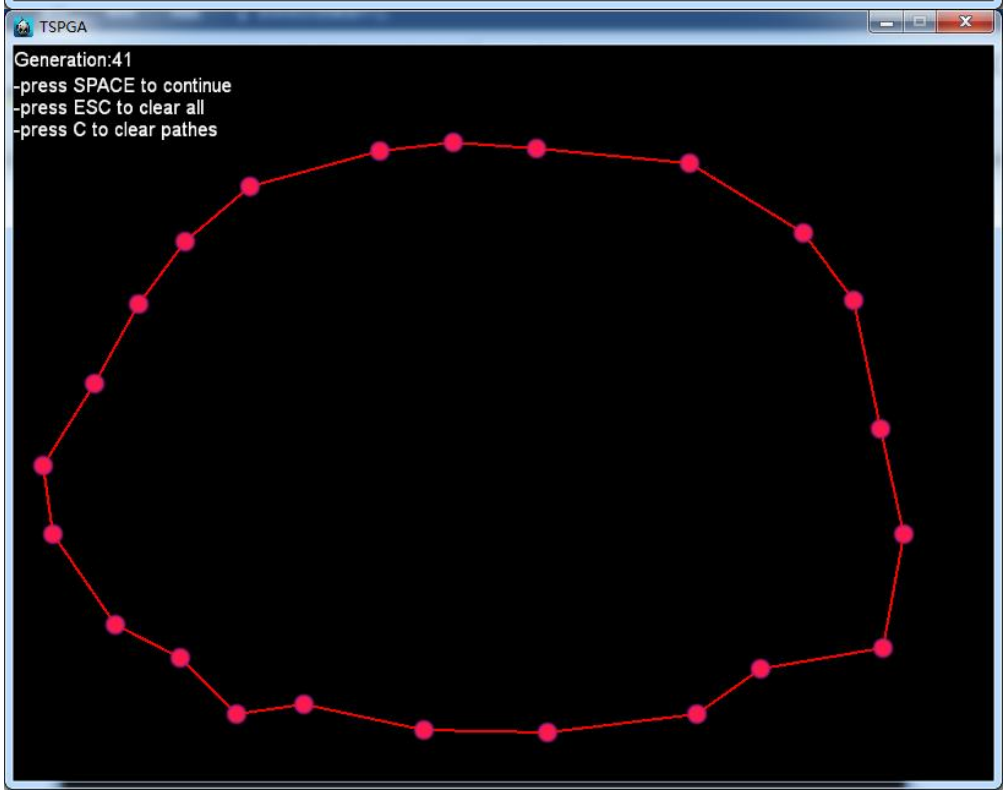
在本项目中，我用一个红点表示一个城市，所以该红点在屏幕上的坐标将用于计算所求封闭回路的长度。在屏幕的任意位置上单击**右键**即可在单击的位置上增加一个红点，表示该处有一座城市。如果觉得点击的位置不恰当，或者想将谋个城市移动到其他位置上去，可以用鼠标**左键**按住谋个红点将其拖动到想要的位置上去。



b.让程序跑起来

创建了足够的城市之后，按**回车键**即可让程序运行起来，这时候将会看到屏幕上有很多杂乱的线不停地跳动，这是因为初始状态下所有的基因都是随机创建的，所以绘制的回路也不固定，多半是一团胡乱的线，但时当程序继续运行的时候，情况开始好转，线路开始变得顺畅起来。最后当线路停止更新时，说明程序已经计算出了最优解，但有时候得到的只是局部最优解，并不是全局最优解，不过大部分时候，程序都能计算出全局最优解。





可以看到，在第 4 代的时候，屏幕上的回路还很杂乱，几乎看不出来是一条经过每个顶点恰好一次的封闭回路。单到了第 41 代的时候，可以明显看到程序已经计算出了一条封闭回路，而且很明显这确实是最短的封闭回路。

c.暂停，继续

在程序运行的过程中，可以按**空格键**暂停遗传算法，然后再按**空格键**可继续执行遗传算法。

d.清除路线，用当前城市重新运行遗传算法

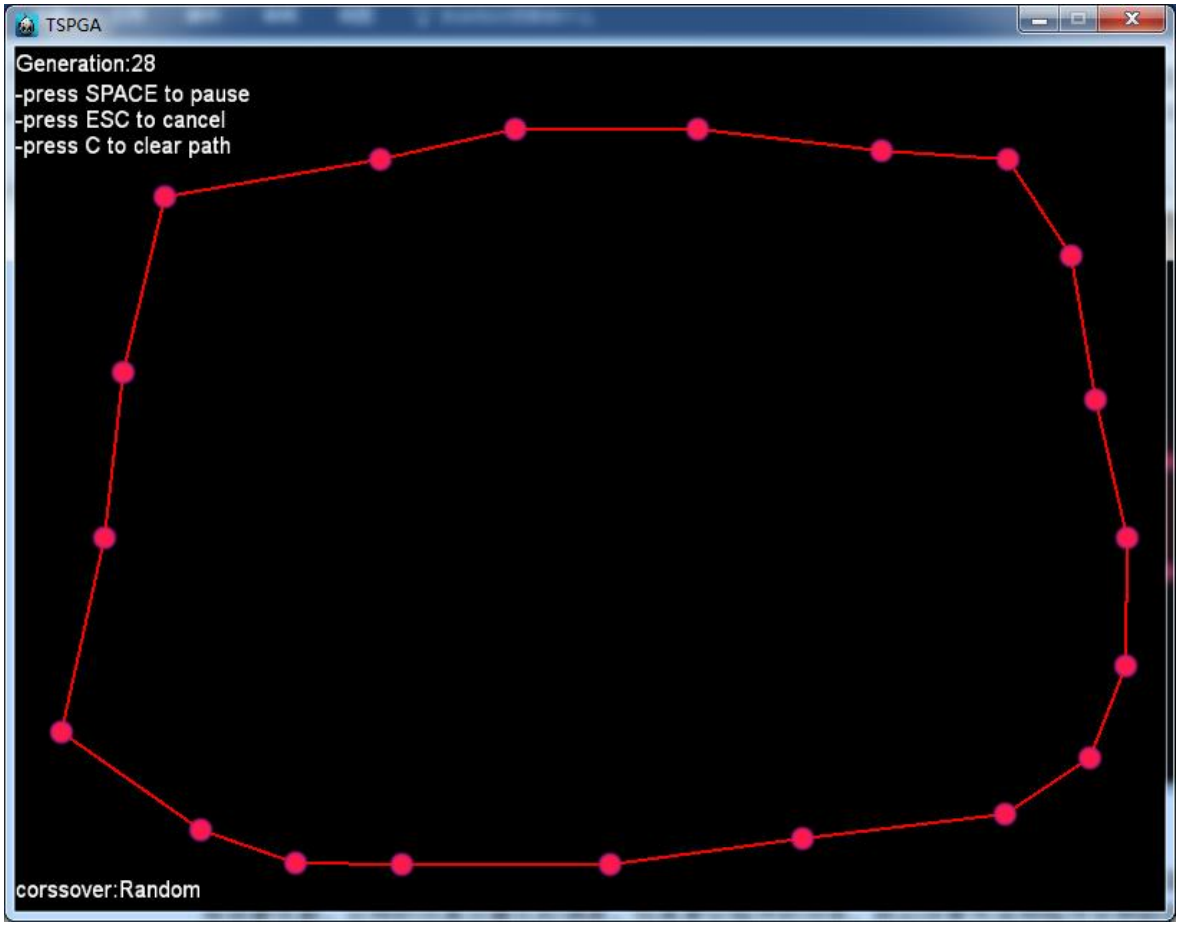
当遗传算法计算出一个全局最优解或者偶尔得到局部最优解时，屏幕上的回路将不再继续更新，这时，可以按 **C 键**清除屏幕上的回路，而所有的城市会保留下来。然后重新按回车键可用当前城市群重新运行遗传算法。由于遗传算法运行过程中具有很大的随机性，因此每一次执行的过程都不一样。实际上，在运行过程中也可以按 C 键清除回路。

e.清除线路和所有的城市，重新创建一群城市

如果觉得当前创建的城市群数量或者摆放的位置不够好，可以按 **ESC 键**清除屏幕上所有的城市，如果屏幕上存在回路也会被清除。然后就可以继续用鼠标右键创建一群城市，用重现创建的城市群来重现运行遗传算法。

f.选择杂交方法

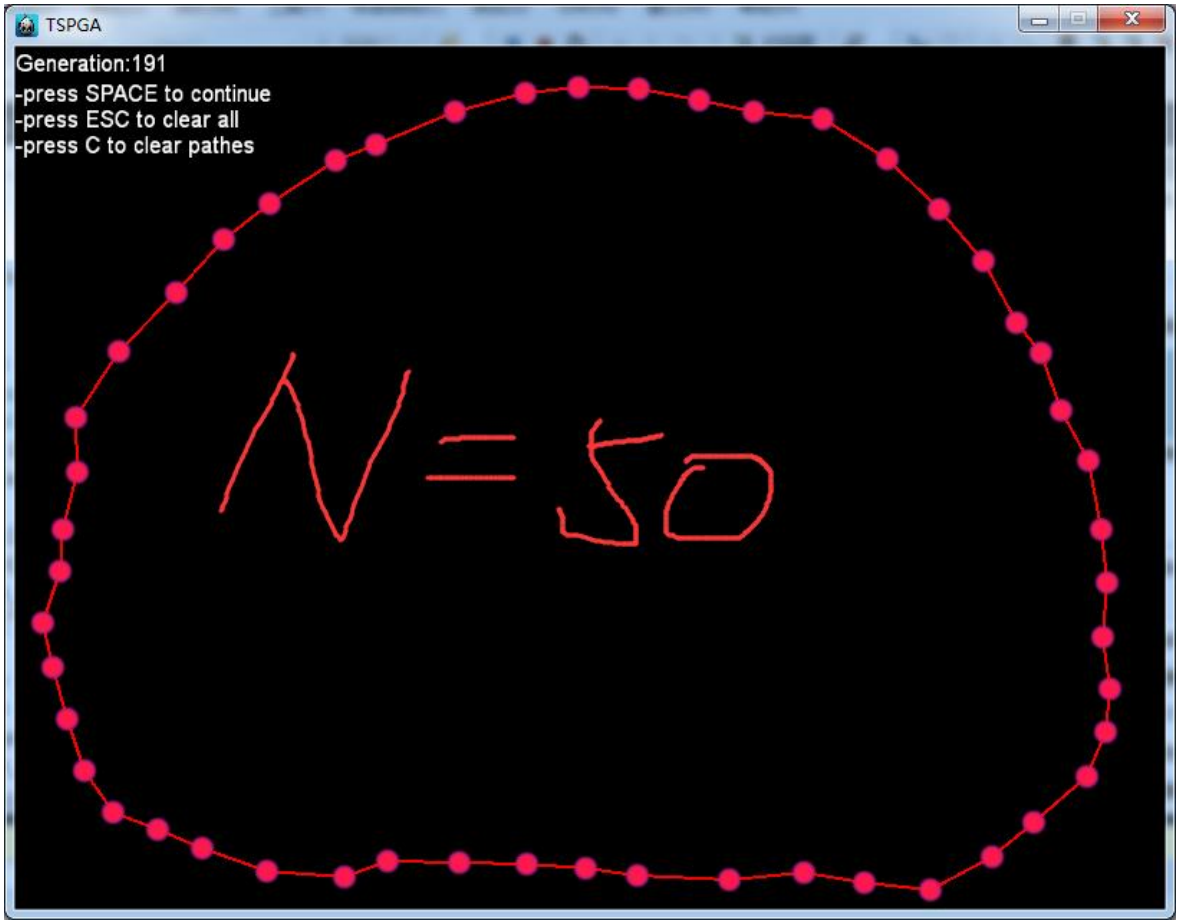
在屏幕左下角有一个按钮，可以选择亲代基因杂交的方法，让选择 Random 时，算法将会在已经的杂交方法中随机选择一种方法进行杂交。切换不同的杂交方法，可以比较不同杂交方法的效果。



五. 算法效果与效率分析

经过测试，在 N 为 20 时，程序能够以 90%的概率计算出全局最优解，会有 10%的概率得到某个局部最优解。这样的效果还算比较满意。如果要谈程序的效率，就必须要先说明程序中种群的规模大小，程序的计算量跟种群规模是成正相关的。在本项目中，设定了种群的规模为城市数量的 15 倍。也就是说，如果有 20 个城市，那么种群中将会有 300 个个体。在这样的规模下，程序一般都能够在 10 秒左右停止更新回路。当 N 为 40 时，在 20 秒以内基本上能得到结果。

以下为实验截图：



六. 何时停止计算

用遗传算法有时候会存在这样一个问题，就是当事先不知道最优解的结果时，程序在运行的时候就不同通过判断是否已经计算出最优解而停止下来，因此必须寻找其他的方法来让算法停止计算，否则算法将永远运行下去。TSP 问题中就存在这一问题，我们事先无法知道最优解是怎样的。下面是采取的一些措施：

```
//  if (fabs(md_bestFitness - md_aveFitness) < md_totalFitness * 0.2) mb_running = false;
//  if (md_sigma / mv_popu.size() < 3000.0) mb_running = false;
```

1. 当最高适应度得分与平均得分的差值小于总得分的 20%时，停止计算
2. 当适应度得分的标准差(md_sigma) 除以种群规模的结果小于 3000 时，停止计算。

但由于遗传算法运行过程中的随机性太大，这两种措施效果都不太好，有时会让算法提前终止，所以在本项目中并没有采用这两种方法。而是采用了下面这种方法：

3.在算法运行的过程中始终记录当前的最优解，如果运行过程中连续 n 代的最优解都没有变，说明算法很可能已经计算出了一个全局最优解或者局部最优解，这时算法将会自动停止计算(这时通

过将成员变量 mb_running 设为 false 实现的)。具体做法为，在程序中设置一个计数器 mi_cntBestTime，如果本代计算出的最优解和上一代计算出的最优解相同，计数器就加 1，如果不同， 则计数器清零。当计数器大于某一个值 mi_maxBestTime 时，算法就将成员变量 mb_running 设为 false 让算法停止计算。这里关键在于如何确定 mi_maxBestTime 的大小，当 mi_maxBestTime 太小时，算法也容易提前终止，当该值设置得太大时，又不能准确反映出算法具体运行了多少代才计算出结果。经过多次测试，本项目中将该值设为了城市数量的 0.5 倍，实验证明这样做能再算计停止更新路线时继续运行数代后停止下来。

```
mi_maxBestTime = mp_map->numberOfCities() * 0.5;

int best = 0;
for (int i = 1,m= src.size(); i < m; ++i) {
    if (src[best].md_fitness < src[i].md_fitness) best = i;
}

if (mg_bestGene == src[best]) {
    if (++mi_cntBestTime >= mi_maxBestTime) {
        mb_running = false;
        mi_cntBestTime = 0;
    }
}
else {
    mi_cntBestTime = 0;
    mg_bestGene = src[best];

}
```

七. 总结

本项目参考了《游戏编程中的人工智能技术》一书，但本项目也有一些创新的地方，例如：本项目中的城市的数量和位置都是可以自由创建的，用户可以创建出任何自己想要的城市群的样子。另外，本项目中采用的杂交方法，选择方法，突变方法以及排名变比方法都是在已有的几种中随机选择使用，这就更加增大了算法的随机性，进一步减小算法进入局部最优解的可能性。

2017 年 12 月 14 日