中国地质大学(武汉)

算法设计与分析课内研讨报告

组　　长：孙鹤轩

组　　员：王诗敏、周迅豪、张书通

指导老师：陈晓宇

目　录

1. 遗传算法描述

遗传算法是模拟生物在自然环境中的遗传和进化过程而形成的一种自适应全局优化概率搜索算法。它最早由美国密执安大学的 Holland 教授提出，起源于 60 年代对自然和人工自适应系统的研究。70 年代 De Jong 基于遗传算法的思想在计算机上进行了大量的纯数值函数优化计算实验。在一系列研究工作基础上，80 年代由 Goldberg 进行工作总结，形成了遗传算法的基本框架。

近年来 ,遗传算法 (GA)的卓越性能引起人们的关注 .对于以往难以解决的函数优化问题 ,复杂的多目标规划问题 ,工农业生产中的配管、配线问题 ,以及机器学习 ,图象识别 ,人工神经网络的权系数调整和网络构造等问题 ,GA 是最有效的方法之一 .虽然 GA 在许多优化问题中都有成功的应用 ,但其本身也存在一些不足 .例如局部搜索能力差、存在未成熟收敛和随机漫游等现象 ,从而导致算法的收敛性能差 ,需要很长时间才能找到最优解 ,这些不足阻碍了遗传算法的推广应用 .如何改善遗传算法的搜索能力和提高算法的收敛速度 ,使其更好地应用于实际问题的解决中 ,是各国学者一直探索的一个主要课题.之后世界范围内掀起了关于遗传算法的研究与应用热潮

2. 遗传算法基本思路

本次研讨主要针对应用遗传算法解决 0-1 背包问题和 TSP 旅行商问题,其中背包问题是重点研究问题。

对于背包问题：0-1 背包是一种组合问题，对于每个物品只有两种状态所以在编码过程中可以采用二进制 0-1 编码，这天然适合染色体结构，其中交叉算子选择单点交叉，变异算子选择逆转基因序列变异，选择算子使用轮盘赌法。

整体算法流程为：

(1) 初始化个体编码

(2) 初始化种群

(3) 染色体修复

(4) 计算适应度

(5) 选择

(6) 交叉

(7) 变异

(8) 重组

(9) 是否满足迭代次数,否转(3)，是算法结束，返回最优解

对于 TSP 问题：编码选择整数编码，设计适应度函数为路径代价的倒数。

3. 算法详细设计

3.1 轮盘赌法选择算子详细设计

表 1 轮盘赌法选择算法基本流程

算法实现 3.1

首先定义选择的个体数量 SELECT_SIZE，接着计算群体适应度之和，开始遍历，取 0-1 间随机数，计算个体被选择的概率，如果满足概率分布则将个体加入选择种群。

伪代码如下：

```
Define SELECT_SIZE  //选择的个体数量
Define Sel_Population[SELECT_SIZE]  //选择种群数组
Produce select(size , pop_size )
    double total_fitness = 0.0;
    for i <- 0 to pop_size do
       total_fitness += population[i].fitness  //计算种群适应度之和
    repeat
    for j <- 0 to size do
       double p = rand() //取 0-1 间随机数
       double sum = 0.0  //轮盘概率区间右区间的值
       for k <- 0 to pop_size do
          sum += population[k].fitness  //将个体的被选择概率相加直到达到被选择
                                        概率
          if sum > p do
            Sel_Population.Add(population[k])
            break
          endif
       repeat
    repeat
end select
```

轮盘赌法的概率选择就是将种群个体根据适应度划分到区间[0 1]，取随机数落到 0-1 区间中，这个随机数作为选择区间的左边界，例如区间划分为[0 0.1],(0.1 0.3],(0.3 0.4],(0.4 0.7],(0.7 1],随机数 p = 0.45,则选择在区间(0.4 0.7]间的个体

表 2 算法 1 核心代码

算法实现 3.1

```
// 选择阶段（轮盘赌法）
            vector<Individual> sel_population;
            double total_fitness = 0.0;
            for (int i = 0; i < POP_SIZE; i++)//计算种群适应度之和
            {
                total_fitness += population[i].fitness;
            }
```

```
        for (int i = 0; i < SELECT_SIZE; i++)//选择SELECT_SIZE个个体
        {
            double p = (double)rand() / RAND_MAX;//取0-1间随机数
            double sum = 0.0;
            for (int j = 0; j < POP_SIZE; j++)
            {
                sum += population[j].fitness / total_fitness;//计算个体被选择的概率
                if (sum >= p)//满足概率分布进行选择操作
                {
                    sel_population.push_back(population[j]);
                    break;
                }
            }
        }
```

3.2 染色体修复（约束处理）

算法实现 3.2

对染色体进行约束处理（将不符合背包容量约束的个体进行更改）以加快收敛速率

算法如下：

（1）将已经装进背包中的物品按照性价比（性价比=价值/质量）由低到高排序；

（2）按照（1）步排序步骤，取走排在第一位的物品，检验背包中剩下的是否满足载重约束。如果满足，则将染色体中基因位上的数字 1 改为 0，此时染色体修复完毕；如果不满足约束，先将染色体中基因位上的数字 1 改为 0，然后取走排在第 2 位的物品，再次检验此时染色体中是否满足背包的载重量约束。循环往复，一直到满足背包的载重量约束位置，染色体初步修复完成。

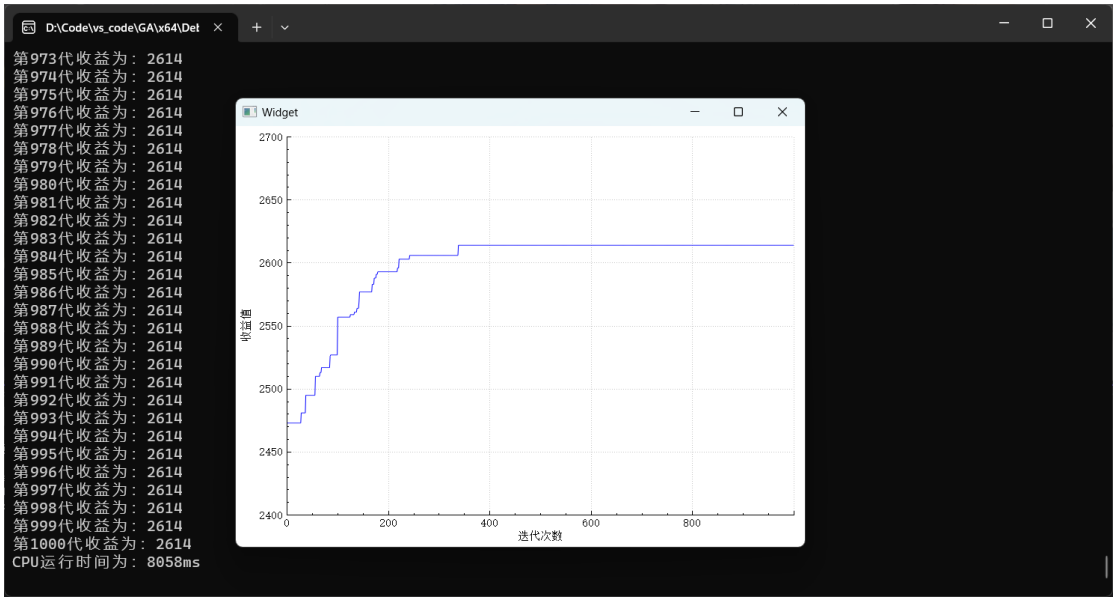
（3）在第（2）步已经满足背包载重量约束的染色体，但此时背包可能还有剩余空间。这时，将此时未装包的物品按照性价比从高到低排序，然后按照该顺序依次将物品装进背包中。在装包过程中，将不满足约束的物品不装包，将满足约束的物品装包，并将染色体基因位上的数字 0 改为数字 1，一直遍历到最后一个未装包的物品为止，染色体修复完毕。

算法实现 3.2

```
//染色体修复,约束处理
    void repair_gene(const vector<Item>& items, int max_weight)
    {
        if (isLegal(items, max_weight))
            return;
        unordered_map<Item, int , hash_name> index;//维护下标的Hash表
        vector<Item> temp1;
```

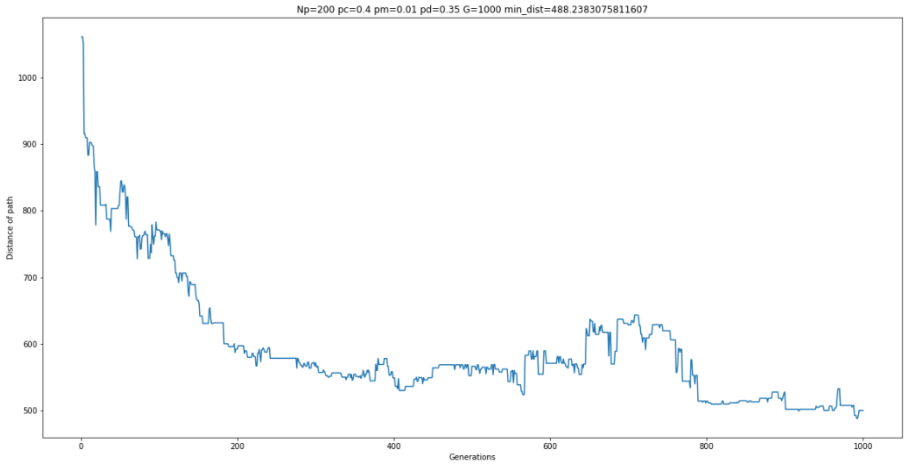
```
        vector<Item> temp2;//用来排序已经放入背包的性价比
        for (int i = 0; i < genes.size(); i++)
        {
            if (genes[i])
            {
                temp1.push_back(items[i]);
                index.emplace(items[i], i);
            }

            else
            {
                temp2.push_back(items[i]);
                index.emplace(items[i], i);
            }
        }
        sort(temp1.begin(), temp1.end());
        for (int i = 0; i < temp1.size(); i++)
        {
            if (isLegal(items, max_weight))
            {
                break;
            }
            int idx = index[temp1[i]];
            genes[idx] = 0;
        }
        sort(temp2.begin(), temp2.end(), greater<Item>());
        for (int i = 0; i < temp2.size(); i++)
        {
            int idx = index[temp2[i]];
            genes[idx] = 1;
            if (isLegal(items, max_weight))
            {
                continue;
            }
            else
            {
                genes[idx] = 0;
            }
        }
    }
```

4. 结果展示与分析



1 背包问题遗传算法运行结果

实验数据的最优解为 2614，可以从图表中看出：随着迭代次数的增加，收益值呈递增式上升，并且在 360 代左右达到最优解，并保持最优。



2 TSP 问题遗传算法运行结果

由上图可以看出随着迭代次数的增加，遗传算法得到的解（路径长度）越来越接近最优解

5. 碰到的问题及解决方案

(1) 遗传算法解决背包问题时陷入局部最优，很难接近最优解

解决方案：因为染色体的基因序列是纯随机生成，所以有很大部分基因序列不满足背包承重约束，因此要进行染色体修复，使初代尽量保持合法的基因序列。

(2) 解决问题时过早成熟收敛

解决方案：一种情况是变异概率设置的过大，经过实验，对于本次的实验数据采用 0.0001 的变异概率比较合适，如果设置的过大会导致在迭代过程中丢弃最优解；另一种情况是我设计了一个算法，即在每次迭代过程中保存最优父代，此做法的好处是收敛很快，可以快速接近最优解，经过实验发现比正常迭代早熟 200 代左右，但是这种做法只能接近最优解而大概率达不到最优解。

(3) 遗传算法的编程实现很复杂

在编程过程中发现遗传算法的编程实现很复杂，几种算子的设计，以及调试过程很艰难，几种算子几乎不能单独调试。

(4) 遗传算法只能适应规模不太大的数据

在实验过程中发现，小规模数据实验中，遗传算法运行速度要比传统 DP 要慢，因为设置了固定的迭代次数，所以遗传算法不论遇到多大规模的数据，其仍然会迭代固定次数，会浪费 cpu 性能，在很大规模数据实验中，因为基因序列的加长，在几种算子的运行上的时间消耗大大增加，这也导致遗传算法的运行时间加长。

(5) 在染色体修复中，使用了 HashMap 来加快遍历速度

在染色体修复中要对每个物品按照收益值进行排序，但是在修复操作中需要对物品进行取舍操作，即改变基因位，所以要记录物品在原先个体基因数组中的位置，所以用 HashMap 来记录其下表，需要注意的是要重载 hash 函数来使物品结构体可以作为键值存入 HashMap，HashMap 避免了重复的遍历操作，使此算法的时间复杂度降低为 O(1)，这给了我们很大的启发，一个良好的数据结构对算法的优化帮助非常大

6. 小组分工情况

分工文档		
分工问题	负责人	工作占比
遗传算法应用研讨报告	孙鹤轩	20.00%
背包问题代码实现	孙鹤轩	20.00%
TSP 问题代码实现	张书通	20.00%
汇报 ppt	王诗敏	20.00%
汇报演讲	周迅豪	20.00%