以0-1背包问题为例实现遗传算法

遗传算法的思想来自生物界，其中涉及适者生存的理念。主要的想法[1]是，根据问题的性质设计编码方式得到个体的染色体，之后初始化一群个体做为种群，并考察这个种群对问题的适应性；对问题适应性强的个体进行染色体交换得到新的个体并得到新的种群，再次得到新种群对问题的适应性，如此循环多代之后，便能得到一个近似解。

0-1背包问题：一个容量为W的背包，并给出n个有价值同时具有重量的物品，要求得到背包所能装的最大价值。

对于该问题，所选用的基因是0或1，由长度为n的0-1字符串表示n个物品的选择情况。生成若干的长度为n的0-1字符串做为初始种群，去求解种群在此问题中的适应性（如果某个体能够在背包容量下得到更大的价值，就表示有更强的适应性），按照适应性选择优秀的染色体进行交换得到新的种群，并重复上述操作直到若干代之后。

以下是按照[2]中伪代码的思路对背包问题的求解方式，所用计算机语言为C++。

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <time.h>
#include <stdlib.h>
using namespace std;

const int Weight = 10000; //背包最大容纳重量
const int MaxWeight = 1000; //随机生成的背包重量最大100
const double MaxValue = 100.0; //随机生成的背包价值最大为100.0
const int G = 1000; //得到1000代种群
const int MaxPopulations = 32; //初始种群的大小
const double Pc = 0.9; //选择染色体交换的概率
const double Pm = 0.0075; //染色体变异的可能性

struct Item {
    double value;
    int weight;
}; //背包的数据类型

void Knapsack(int n); //用于求解背包问题
double GA(int n, vector<struct Item> items); //遗传算法
double DP(int n, vector<struct Item> items); //动态规划
void generateItem(int n, vector<struct Item> &items); //生成背包
void initPopulations(int n, vector<string> &populations); //初始化种群
double getFitness(vector<struct Item> items, vector<string>
populations, vector<double> &fitness); //得到各个体的适应度
int selection(vector<double> fitness); //选择染色体交换的个体
void crossover(string &female, string &male, int n); //交换两条染色体
```

```cpp
void mutation(string &individuals, int n); //变异

int main() {
    int n; //背包数量

    srand((unsigned int)time(NULL)); //随机数种子，便于生成不同的随机数
    cout<<"Input number of items:";
    do {
        cin>>n;
    }while(n <= 0); //保证输入的背包个数大于0
    Knapsack(n);

    return 0;
}

void Knapsack(int n) {
    vector<struct Item> items; //定义物品

    generateItem(n, items); //生成物品
    for (int i = 0; i < n; ++i) cout<< items[i].value << " "
<<items[i].weight <<endl; //输出背包
    cout<< GA(n, items) <<endl; //输出用遗传算法得到的近似解
    cout<< DP(n, items) <<endl; //输出用动态规划得到的精确解
}

double GA(int n, vector<struct Item> items) {
    int female, male, descendants = 0; //依次为雌性个体、雄性个体、第几代
    double maxValue = 0; //最大的价值
    string bestIndividual; //最优的个体
    vector<string> populations; //定义种群
    vector<double> fitness; //定义适应度

    for (int i = 0; i < MaxPopulations; ++i) fitness.push_back(0); //
初始化适应度
    initPopulations(n, populations); //初始化种群

    do {
        descendants += 1; //记录第几代
        maxValue = max(maxValue, getFitness(items, populations,
fitness)); //得到最大价值
        for (int i = 0; i < MaxPopulations; ++i) { //对种群进行选择、交
叉、变异操作的遗传算子
            if (1.0 * rand() / RAND_MAX < Pc) { //判断是否可以进行选择
                female = selection(fitness); //选择雌性个体
                male = selection(fitness); //选择雄性个体
                crossover(populations[female], populations[male], n);
//染色体交叉
                if (1.0 * rand() / RAND_MAX < Pm) { //判断是否发生变异
                    mutation(populations[female], n);
                    mutation(populations[male], n);
                }
            }
```

```
        }
    }while(descendants < G); //1000代之后结束

    return maxValue; //返回最大价值
//    return bestIndividual; //返回近似解
}

void generateItem(int n, vector<struct Item> &items) {
    struct Item item;

    for (int i = 0; i < n; ++i) {
        item.value = (1.0 * rand() / RAND_MAX) * MaxValue;
        item.weight = rand() % MaxWeight;
        items.push_back(item); //将生成的物品放入待取区
    }
}

void initPopulations(int n, vector<string> &populations) {
    for (int i = 0; i < MaxPopulations; ++i) { //生成种群
        string tmp = "";
        for (int j = 0; j < n; ++j) //随机生成个体
            tmp += rand() % 2 + '0';
        populations.push_back(tmp); //将生成的个体放入种群
    }
}

double getFitness(vector<struct Item> items, vector<string>
populations, vector<double> &fitness) {
    double allValue = 0, maxValue = 0; //分别表示种群的价值总和、最大价值
    string bestIndividual; //保留最优的个体

    for (int i = 0; i < MaxPopulations; ++i) { //对每个个体计算适应度
        double maxWeight = 0; //用于记录个体所需的背包容量
        fitness[i] = 0;
        for (int j = 0; j < populations[i].length(); ++j) {
            maxWeight += (populations[i][j] - '0') * items[j].weight;
            if (maxWeight > Weight) { //当个体所需的容量大于背包容量时，解不
存在
                fitness[i] = 0;
                break;
            }
            fitness[i] += (populations[i][j] - '0') * items[j].value;
        }
        if (fitness[i] > maxValue) { //用于得到最优解
            maxValue = fitness[i];
            bestIndividual = populations[i];
        }
        allValue += fitness[i]; //计算种群价值
    }
    if (allValue == 0) allValue = 1; //防止所有的解都不存在
    for (int i = 0; i < MaxPopulations; ++i) //求解适应度，价值大的适应度
大
```

```
125         fitness[i] /= allValue;
126     return maxValue; //返回最大价值
127 //    return bestIndividual; //返回选择方式
128 }
129 
130 int selection(vector<double> fitness) {
131     double m = 0;
132     double r = 1.0  * rand() / RAND_MAX; //随机生成需要选择的个体
133 
134     for (int i = 0; i < MaxPopulations; ++i) { //类似转动圆盘，最后选择指针所指的区域
135         m += fitness[i];
136         if (r <= m) return i; //返回之后指针
137     }
138     return MaxPopulations - 1; //当所有值都不存在时，保证有值返回
139 }
140 
141 void crossover(string &female, string &male, int n) {
142     string subOfFemale = female.substr(n/4, n/2); //选择female中n/4-n/2之间的片段
143     string subOfMale = male.substr(n/4, n/2); //选择male中n/4-n/2之间的片段
144 
145     female.replace(n/4, subOfFemale.length(), subOfMale); //将male中的染色体放入female中
146     male.replace(n/4, subOfMale.length(), subOfFemale); //将female中的染色体放入male中
147 }
148 
149 void mutation(string &individuals, int n) {
150     int index = rand() % n; //随机某个基因位
151 
152     individuals[index] = !(individuals[index] - '0') + '0'; //更改基因
153 }
154 
155 double DP(int n, vector<struct Item> items) {
156     vector<double> value[Weight + 1];
157 
158     for (int i = 0; i < Weight + 1; ++i) //得到存储解的表
159         for (int j = 0; j < n + 1; ++j)
160             value[i].push_back(0);
161 
162     for (int i = 0; i < n + 1; ++i) value[0][i] = 0; //初始化
163     for (int i = 0; i < Weight + 1; ++i) value[i][0] = 0; //初始化
164 
165     for (int i = 1; i < n + 1; ++i) { //用动态规划求解
166         for (int j = 1; j < Weight + 1; ++j) { //以下时状态转移方程
167             if (items[i - 1].weight > j) value[j][i] = value[j][i - 1];
168             else value[j][i] = max(value[j][i - 1], value[j - items[i - 1].weight][i - 1] + items[i - 1].value);
169         }
```

```
    }
    return value[Weight][n]; //返回最优值
}
/*
 1. 物品数量和种群数量之间的关系对解的影响
 2. 多少代种群对解的影响
*/

```

但是当输入物品的规模大于种群数量时，近似效果并不理想。考虑到可能是生成的个体对于问题来说都是无解，即所需的容量远远大于背包提供的容量。于是尝试更改随机生成个体的方式，假设用均值的重量放入背包中（也许采用中位数效果可能更好，然而还剩三天要期末考试了😂），则可放入的物品数为$m = \frac{Weight}{meanWeight}$，然后对$m$个基因位赋值为1，其余为0。

对于新的初始种群的代码如下：

```
void initPopuWithMean(int n, vector<string> &populations, vector<struct Item> items) {
    int index = 0, meanWeight = 0, preChoose;

    for (int i = 0; i < items.size(); ++i)
        meanWeight += items[i].weight;
    meanWeight /= items.size();
    preChoose = meanWeight > n? n: meanWeight; //判断是否可选数目，使得预先选择的想法可以实现
    for (int i = 0; i < MaxPopulations; ++i) { //生成种群
        string tmp(n,'0'); //初始化，预先不取任何物品
        for (int j = 0; j < preChoose; ++j) {//随机生成个体
            index = rand() % n; //选定需要更改的基因位
            tmp[index] = '1';
        }
        populations.push_back(tmp); //将生成的个体放入种群
    }
}
```

初始种群的方式采用上述方式的时候，在物品的数量远大于种群个数时，相对于原先的初始方式出现0的可能性减少很多。另外的优化可以考虑对种群中适应度极低的个体进行淘汰，提高种群的品质，但是要防止种群沉迷于局部最优解。

> 后话：
>
> 由于预想尝试运用关系型数据（如：社交网络，人口迁移）中的有向关系（必要性）对数据进行聚类，发现已建立的模型无法以下几点：
>
> 1. 模型无法判断何时将数据聚合
> 2. 模型是否可以终止聚类算法。
>
> 以至于草草做了遗传算法。😂😅

---

1. 遗传算法GA(Genetic Algorithm)入门知识梳理↩
2. 遗传算法入门 ↩