群体智能优化算法

遗传算法

遗传算法(Genetic Algorithm, GA)是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。

该算法通过数学的方式，利用计算机仿真运算，将问题的求解过程转换成类似生物进化中的染色体基因的**选择**、**克隆**、**交叉**、**变异**等过程。

示例

$$\begin{aligned} \max \quad & f(x_1, x_2, x_3) = -x_1 x_2 - (x2 - 0.05)(x_2 - 0.05) - x_3 x_3 \\ & x_i \in [0, 1], x_2 \in [-1, 1], x_3 \in [0.5, 1] \end{aligned}$$

```
#include "GA.h"
using namespace std;

double func(double x[])
{
    double x1 = x[0];
    double x2 = x[1];
    double x3 = x[2];
    return -x1 * x2 - (x2 - 0.05) * (x2 - 0.05) - x3 * x3;
}

int dim = 3;
double lb[3] = { 0, -1, 0.5 };
double ub[3] = { 1, 1, 1 };

GA ga = GA(func, dim, lb, ub);
//GA ga = GA(func, dim, lb, ub, chrom_len, pop, iter_max, pc, pm)
ga.Optimize();

cout << "x = " << ga.best_x[0] << " " << ga.best_x[1] << " " << ga.best_x[2] <<
endl;
cout << "y = " << ga.best_y << endl;
```

免疫算法

免疫算法(Immune Algorithm, IA)将免疫概念及其理论应用于遗传算法，在保留原算法优良特性的前提下，力图有选择、有目的地利用待求问题中的一些特征信息或知识来抑制其优化过程中出现的退化现象。

对于免疫算法，**抗原**就是待解决的优化问题，**抗体**就是对应问题的解的结构。然后针对抗体种群进行质量评价，评价准则是个体**亲和度**和个体**浓度**，评价得出的优质抗体将进行免疫操作，劣质抗体将会被刷新。 免疫操作：利用免疫**选择**、**克隆**、**变异**、**克隆抑制**等算子模拟生物免疫应答中的各种免疫操作，形成基于生物免疫系统克隆选择原理的进化规则和方法，实现对各种最优化问题的寻优搜索。

抗体**鼓励度**是对抗体质量的最终评价结果，通常亲和度大，浓度低的抗体会得到较大的鼓励度。

$$\begin{aligned} & enc(s) = aff(s) - w \cdot sim(s) \\ \text{or} \quad & enc = aff(s) e^{-w \cdot sim(s)} \end{aligned}$$

示例

```
#include "IA.h"

IA ia = IA(func, dim, lb, ub);
//IA ia = IA(func, dim, lb, ub, chrom_len, pop, iter_max, pc, pm, ps, ws)
ia.Optimize();

cout << "x = " << ia.best_x[0] << " " << ia.best_x[1] << " " << ia.best_x[2] <<
endl;
cout << "y = " << ia.best_y << endl;
```

差分进化算法

差分进化算法(Differential Evolution Algorithm, DE)是一种高效的全局优化算法。它也是基于群体的启发式搜索算法，群中的每个个体对应一个解向量。

DE算法通过采用浮点矢量进行编码生成种群个体。在DE算法寻优的过程中，首先，从父代个体间选择两个个体进行向量做差生成差分矢量；其次，选择另外一个个体与差分矢量求和生成实验个体；然后，对父代个体与相应的实验个体进行交叉操作，生成新的子代个体；最后在父代个体和子代个体之间进行选择操作，将符合要求的个体保存到下一代群体中去。

示例

```
#include "DE.h"
using namespace de;

auto evo = diff_evo(func, dim, lb, ub);
auto sol = evo.optimize();
y = func(sol);

cout << "optimal solution:" << endl;
for(int i = 0; i < sol.size(); ++i){
    cout << sol[i] << endl;
}
cout << "optimal cost: " << y << endl;
```

模拟退火算法

爬山算法是一种简单的**贪心搜索算法**，该算法每次从当前解的临近解空间中选择一个最优解作为当前解，直到达到一个局部最优解。爬山算法实现很简单，其主要缺点是会陷入局部最优解，而不一定能搜索到全局最优解。

模拟退火算法(Simulated Annealing, SA)基于Monte-Carlo迭代求解策略的一种随机寻优算法。

模拟退火算法从某一较高初温出发，伴随温度参数的不断下降，结合概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。

如果新解比当前解更优，则接受新解，否则基于Metropolis准则判断是否接受新解。接受概率为：

$$P = \begin{cases} 1, & E_{t+1} < E_t \\ e^{-\frac{E_{t+1}-E_t}{kT}}, & E_{t+1} \geq E_t \end{cases}$$

示例

旅行商问题（Traveling Salesman Problem, TSP）：假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

```
#include "SA.h"
#define _CRT_SECURE_NO_WARNINGS 1
using namespace std;

void main()
{
    srand((unsigned)time(NULL));

    time_t start, finish;
    start = clock();

    Simulated_Annealing SA1;
    SA1.in();
    SA1.SA();

    finish = clock();
    double run_time = ((double)(finish - start)) / CLOCKS_PER_SEC;

    printf("模拟退火算法解TSP问题：\n");
    cout << "初始温度T0=" << T0 << ",降温系数q=" << q << endl;
    printf("每个温度迭代%d次,共降温%d次。\n", L, SA1.getCount());

    SA1.get_All_solutions();
    SA1.out();

    printf("最终得TSP问题最优路径为：\n");
    for (int j = 0; j < C - 1; j++)
    {
        printf("%d->", SA1.getCurrent_Solution(j));
    }
    printf("%d->", SA1.getCurrent_Solution(C - 1));
    printf("%d\n", SA1.getCurrent_Solution(0));

    cout << "最优路径长度为：" << SA1.getF1() << endl;
    printf("程序运行耗时%1f秒.\n", run_time);
    system("pause");
}
```

禁忌搜索算法

禁忌搜索(Tabu Search, TS)是一种亚启发式随机搜索算法，它从一个初始可行解出发，选择一系列的特定搜索方向（邻域）作为试探，选择实现让特定的目标函数值变化最多的移动。

为了避免陷入局部最优解，禁忌搜索中采用了一种灵活的"记忆"技术，对已经进行的优化过程进行记录和选择，指导下一步的搜索方向，这就是**禁忌表**的建立。为了达到全局最优，我们会让一些禁忌对象重新可选，这种方法称为**特赦**，相应的规则称为特赦规则。

示例

```
#include "../src/TS.h"

using namespace std;
```

```
double HeuristicValue(double* x, double* y)
{
    return sqrt((x[0] - y[0]) * (x[0] - y[0]) + (x[1] - y[1]) * (x[1] - y[1]));
}

int main()
{
    srand((int)time(0));

    TS ts = TS(HeuristicValue, n_cands, n_cities);
    //TS ts = TS(HeuristicValue, n_cands, n_cities, iter_max);

    for (int i = 0; i < n_cities; i++)
    {
        x = rand() / double(RAND_MAX);
        y = rand() / double(RAND_MAX);
        ts.CreateCities(i, x, y);
    }

    ts.Optimize();

    cout << endl << "Best route:" << endl;
    for (int i = 0; i < n_cities; i++)
        cout << ts.route_best[i] << " ";
    cout << endl << "Best reward:" << endl << ts.reward_best << endl;
}
```

粒子群算法

粒子群优化算法(Particle Swarm Optimization, PSO)是通过模拟鸟群觅食行为而发展起来的一种基于群体协作的随机搜索算法。

PSO中，每个优化问题的解都是搜索空间中的一只鸟，我们称之为“粒子”。所有的粒子都有一个由被优化的函数决定的适应值，每个粒子还有一个速度决定他们飞翔的方向和距离。然后粒子们就追随当前的最优粒子在解空间中搜索。

速度及位置更新公式：

$$v_{k+1} = \omega v_k + c_1 r_1 \left(x_{pbest,k} - x_k\right) + c_2 r_2 \left(x_{gbest,k} - x_k\right)$$

$$x_{k+1} = x_k + v_{k+1}$$

式中，w为惯性系数，c_1，c_2为权重，r_1，r_2为随机数，$pbest$表示单个粒子的历史最优解，$gbest$表示所以粒子的最优解。

示例

```
#include "PSO.h"

PSO pso = PSO(func, dim, lb, ub);
//PSO pso = PSO(func, dim, lb, ub, pop, iter_max, w, cp, cg)
pso.Optimize();

cout << "x = " << pso.gbest_x[0] << " " << pso.gbest_x[1] << " " <<
pso.gbest_x[2] << endl;
cout << "y = " << pso.gbest_y << endl;
```

蚁群算法

蚁群算法(Ant Colony Optimization, ACO)是一种用来寻找优化路径的概率型算法。

将蚁群算法应用于解决优化问题的基本思路为：用蚂蚁的行走路径表示待优化问题的可行解，整个蚂蚁群体的所有路径构成待优化问题的解空间。路径较短的蚂蚁释放的信息素量较多，随着时间的推进，较短的路径上累积的信息素浓度逐渐增高，选择该路径的蚂蚁个数也愈来愈多。最终，整个蚂蚁会在正反馈的作用下集中到最佳的路径上，此时对应的便是待优化问题的最优解。

转移概率为：

$$p_{ij}^{k}(s)=\begin{cases}\dfrac{\tau_{ij}(s)^{\alpha}\eta_{ij}(s)^{\beta}}{\sum\limits_{l\in allow_k}\tau_{il}(s)^{\alpha}\eta_{il}(s)^{\beta}}, & j\in allow_k\\ 0, & j\notin allow_k\end{cases}$$

信息素浓度更新公式为：

$$\begin{cases}\tau_{ij}(t+1)=(1-\rho)\tau(s)+\Delta\tau_{ij}\\ \Delta\tau_{ij}=\sum\limits_{k=1}^{m}\Delta\tau_{ij}^{k}\end{cases}$$

蚁周模型(ANT-cycle)：信息素增量$\Delta\tau^{k}=\dfrac{Q}{l_k}$，与全局路径总距离相关；

蚁量模型(ANT-quantity)：信息素增量$\Delta\tau_{ij}=\dfrac{Q}{d_{ij}}$，与局部路径距离相关；

蚁周模型(ANT-cycle)：信息素增量$\Delta\tau=Q$，与路径距离无关。

式中，$allow$表示可访问城市列表，η为启发函数，α，β分别为信息素浓度和启发值的权重，Q为蚂蚁携带信息素总量，ρ为信息素挥发因子。

示例

```
#include "../src/TS.h"

int main()
{
    srand((int)time(0));

    ACO aco = ACO(HeuristicValue, n_ants, n_cities, iter_max, alpha, beta, q,
rho, tau_max);
    //ACO aco = ACO(HeuristicValue, n_ants, n_cities);

    for (int i = 0; i < n_cities; i++)
    {
        x = rand() / double(RAND_MAX);
        y = rand() / double(RAND_MAX);
        aco.CreateCities(i, x, y);
    }

    aco.Optimize();

    cout << endl << "Best route:" << endl;
    for (int i = 0; i < n_cities; i++)
        cout << aco.route_best[i] << " ";
    cout << endl << "Best reward:" << endl << aco.reward_best << endl;
```

```
}
```

蒙特卡洛树搜索

蒙特卡洛树搜索(Monte Carlo tree search, MCTS)是一种用于某些决策过程的启发式搜索算法，

在理想状态下，应当选择胜率最高的子节点进行搜索。但是，只有有了足够多的搜索次数，才能较为准确地估计每个节点的胜率。于是，选择子节点时要在开发（搜索被访问次数较少的节点）与利用（选择胜率高的节点）之间进行权衡。根据前人的研究，一般会基于如下规则选择要访问的子节点：

1. 如果存在没有被访问过的子节点，在这些节点中随机选择一个；
2. 如果所有子节点都被访问过，根据下面这个公式选择要移动到的子节点：

$$UCT(v) = \frac{w_v}{n_v} + c\sqrt{\frac{\ln n_u}{n_v}}$$

其中，w_v 表示节点 v 对于节点u而言 的胜利次数，n_x 表示节点 x 被搜索的次数，而 c 是一个常数，一般取 2 左右，可以根据实际情况调参。

这个公式的意义在于：前半部分是胜率，而后半部分是一个访问次数越少值越大的函数。于是，选择 UCT 最高的子节点，就可以做好开发与利用之间的平衡。

示例

```
#include "../src/MCTS.h"

int main()
{
    srand((int)time(0));

    MCTS mcts = MCTS(HeuristicValue, n_cities);
    //MCTS mcts = MCTS(HeuristicValue, n_cities, iter_max, cp);

    for (int i = 0; i < n_cities; i++)
    {
        x = rand() / double(RAND_MAX);
        y = rand() / double(RAND_MAX);
        mcts.CreateCities(i, x, y);
    }

    mcts.Optimize();

    cout << endl << "Best route:" << endl;
    for (int i = 0; i < n_cities; i++)
        cout << mcts.route_best[i] << " ";
    cout << endl << "Best reward:" << endl << mcts.reward_best << endl;
}
```

BP算法

人工神经网络（Artificial Neural Networks，ANN）是由众多的神经元可调的连接权值连接而成，具有大规模并行处理、分布式信息存储、良好的自组织自学习能力等特点。

BP（Error Back Propagation）算法又称为误差反向传播算法，是人工神经网络中的一种监督式的学习算法。BP 神经网络算法在理论上可以逼近任意函数，基本的结构由非线性变化单元组成，具有很强的非线性映射能力。而且网络的中间层数、各层的处理单元数及网络的学习系数等参数可根据具体情况设定。

示例

```
#include "../src/BP.h"

int main()
{
    BpNet testNet;

    // 学习样本
    vector<double> samplein[4];
    vector<double> sampleout[4];
    samplein[0].push_back(0); samplein[0].push_back(0);
sampleout[0].push_back(0);
    samplein[1].push_back(0); samplein[1].push_back(1);
sampleout[1].push_back(1);
    samplein[2].push_back(1); samplein[2].push_back(0);
sampleout[2].push_back(1);
    samplein[3].push_back(1); samplein[3].push_back(1);
sampleout[3].push_back(0);
    sample sampleInOut[4];
    for (int i = 0; i < 4; i++)
    {
        sampleInOut[i].in = samplein[i];
        sampleInOut[i].out = sampleout[i];
    }
    vector<sample> sampleGroup(sampleInOut, sampleInOut + 4);
    testNet.training(sampleGroup, 0.0001);

    // 测试数据
    vector<double> testin[4];
    vector<double> testout[4];
    testin[0].push_back(0.1);   testin[0].push_back(0.2);
    testin[1].push_back(0.15);  testin[1].push_back(0.9);
    testin[2].push_back(1.1);   testin[2].push_back(0.01);
    testin[3].push_back(0.88);  testin[3].push_back(1.03);
    sample testInOut[4];
    for (int i = 0; i < 4; i++) testInOut[i].in = testin[i];
    vector<sample> testGroup(testInOut, testInOut + 4);

    // 预测测试数据，并输出结果
    testNet.predict(testGroup);
    for (int i = 0; i < testGroup.size(); i++)
    {
        for (int j = 0; j < testGroup[i].in.size(); j++) cout <<
testGroup[i].in[j] << "\t";
        cout << "-- prediction :";
        for (int j = 0; j < testGroup[i].out.size(); j++) cout <<
testGroup[i].out[j] << "\t";
        cout << endl;
    }

    system("pause");
```

```
    return 0;
}
```

参考文献

@inproceedings{Eberhart2002,
  author = {Eberhart, R. and Kennedy, J.},
  title = {A new optimizer using particle swarm theory},
  booktitle = {Mhs95 Sixth International Symposium on Micro Machine & Human Science},
  pages = {39-43},
  year = {2002},
  type = {Conference Proceedings}
}

@inproceedings{Colorni1991,
  author = {Colorni, A. and Dorigo, M. and Maniezzo, V.},
  title = {Distributed optimization by ant colonies},
  booktitle = {Proc of the First European Conference on Artificial Life},
  pages = {134-142},
  year = {1991},
  type = {Conference Proceedings}
}