Algorithm

1 算法概述

1.1 算法基本概念

- 算法是一系列解决问题的清晰指令。对于符合一定规范的输入，算法能够在有限时间内获得所要求的输出
- 算法是解决问题的一种方法或过程，它是由若干条指令组成的有穷序列

一个算法应该具有以下几个方面的特征：

- **输入**：有零或多个外部量作为算法的输入
- **输出**：算法产生至少一个量作为输出
- **确定性**：组成算法的每条指令清晰、无歧义
- **有效性**：算法中执行的任何计算步骤都可以被分解为基本的可执行的操作步
- **有限性**：算法中每条指令的执行次数有限，执行每条指令的时间也有限

1.2 算法求解问题基本过程

- 算法设计：算法思想；算法描述（自然语言、流程图、伪代码……）
- 算法的正确性证明：归纳法、反证法……
- 算法分析：正确性，效率，简单性，一般性
- 算法的程序实现

2 算法效率分析基础

2.1 算法效率度量

算法效率的高低体现在运行该算法所需要耗费资源的多少，对于计算机来讲，最重要的资源是时间和空间，因此，算法效率又可分为**时间效率**和**空间效率**

N：要解决问题的规模；I：算法的输入；A：算法本身

复杂性：$C = F(N, I, A)$

将复杂性分开：时间复杂性 $T=F(N,I,A)$，空间复杂性 $S=F(N,I,A)$。

$T=T(N,I)$， $S=S(N,I)$

5 动态规划

步骤：

1. 找出最优子结构
2. 建立递推关系式
3. 计算最优值
4. 构造最优解

6 贪心算法

应用贪心算法求解问题的关键在于**贪心策略的选择**。

最小生成树：Prim

先找任一结点，然后每次找周边最小边。$O(n^2)$

最小生成树： Kruskal

先找最短边，然后依次找最小边，同时避免生成环。$O(eloge)$

边数较少时Prim算法效率较高

单源最短路径问题：Dijkstra

1. 初始化s[], dist[], prev[]
2. 找到新的最近s外结点，将其添加到s[]
3. 更新加入新结点后的dist[], prev[]

```
template <class Type>

void Dijkstra(int n, int v, Type dist[], int[] prev, Type** c) {
    // n节点数，v源结点，dist特殊路径，prev前驱结点矩阵，c邻接矩阵（表示结点间距离）
    bool s[maxint];
    for (int i = 1; i <= n; i++) { // 遍历所有节点
        s[i] = false; // 未被选择（初始化s[]）
```

```
        dist[i] = c[v][i]; // 初始化dist[]: 结点v到结点i的距离（无则为maxint)
        if (dist[i] == maxint) { //若v和i结点无直接连接
            prev[i] = 0;
        } else {
            prev[i] = v; // i结点的前驱为v
        }
    }
    dist[v] = 0; // (初始化dist: 源结点距离为0)
    s[v] = true; // 源结点已被选中

    for (int i = 1; i < n; i++) { // n-1?
        int temp = maxint;
        int u = v; // 路径起点
        for (int j = 1; j <= n; j++) { // 选出结点
            if ((!s[j]) && (dist[j] < temp)) { // j未被选中 且 已选中结点到j的特殊路径距离存在
                u = j;
                temp = dist[j]; // 最终temp为最小值，u为对应的结点
            }
        }
        s[u] = true; // 已选中u（上述循环中满足特殊路径最小的j)
        for (int j = 1; j <= n; j++) {
            if ((!s[j]) && (c[u][j]) < maxint) { // 已选中结点的相邻未选中结点
                Type newdist = dist[u] + c[u][j]; // 源→u + u→j
                if (newdist < dist[j]) {
                    dist[j] = newdist;
                    prev[j] = u;
                }
            }
        }
    }
}
```

7 回溯算法

也称“试探法”。

搜索策略：**深度优先**为主，也可以采用广度优先、函数优先、广度深度结合等。

避免无效搜索策略：
约束函数：在扩展结点处剪去不满足约束条件的子树
界限函数：在扩展结点处剪去得不到最优解的子树

算法框架

递归回溯

```
/*
    t: 递归深度; n: 最大递归深度; x : 解向量
    output(x): 输出得到的可行解x;
    f(n, t),  g(n, t): 当前扩展结点处未搜索的子树的起始编号和终止编号;
    h(i): 当前扩展结点x[t]的第i个可选值;
    constraint(t),  bound(t): 当前扩展结点的约束函数, 界函数;
*/
void backtrack(int i) {
    if (i > n) output(x);
    else
        for (int i = f(n, f); i ≤ g(n, t); i++) {
            x[t] = h(i);
            if (constraint(t) && bound(t))
                backtrack(t + 1);
        }
}
```

迭代回溯

```
/*
    t: 迭代深度; x : 解向量;
    output(x): 输出得到的可行解x;
    f(n, t),  g(n, t): 当前扩展结点处未搜索的子树的起始编号和终止编号;
    h(i): 当前扩展结点x[t]的第i个可选值;
    constraint(t),  bound(t): 当前扩展结点的约束函数, 界函数;
    solution(t): 判断当前扩展结点处是否得到问题的可行解
*/
void iterativeBacktrack ()
{
    int t = 1;
    while (t > 0) { //退到无路可退, 结束
        if (f(n, t) ≤ g(n, t))
            for (int i = f(n, t); i ≤ g(n, t); i++) {
```

```
                x[t] = h(i);
                if (constraint(t) && bound(t)) {
                    if (solution(t)) output(x);
                    else t++; //等价于backtrack(t+1)
                }
            }
        else t--;//走不通了后退
    }
}
```

N皇后问题

1：8*8棋盘，此算法不适用于规模n的问题

```
void Queen1() {
    int x[9]; // 存储各皇后的位置
    for x[1] = 1:8
        for x[2] = 1:8
            if(check(x, 2) = 0) continue;
            for x[3] = 1: 8
                if(check(x, 3) = 0) continue;
               ...
                for x[8] = 1: 8
                    if(check(x, 8) = 0) continue;
                    else print(x);
}

int check(int x[], int pos) { // 检查第pos个皇后是否与之前的冲突
    for (int i = 1; i < pos; i++) {
        if (abs(x[i]-x[pos])==abs(i-pos) or x[i]==x[pos]) // 对角线 or 同列
            return 0;
    }
    return 1;
}
```

2：递归回溯

```
int n = *;
sum = 0;
x[n];
void Backtrack(int t) {
```

```
    if (t > n) sum++;
    else
        for (int i = 1; i <= n; i++) {
            x[t] = i;
            if (check(x, t))
                Backtrack(t + 1);
        }
}

int check(int x[], int pos) {
    for(int i = 1; i < pos; i++)
        if(abs(x[i] - x[pos]) == abs (i - pos) or x[i] == x[pos])
            return 0;
    return 1;
}
```

3：迭代求解（？？？）

```
void iterativeBacktrack (int n)
{
    int t = 1;
    int[] f;// 记录每一层皇后尝试到哪了
    while (t > 0) {
        if (f[t] <= n)
            for (; f[t] <= n; f[t]++) {
                x[t] = f[t];
                if(check(x, t))
                    if (t = n) output(x);
                else
                    t++;
            }
        else {
            t--;
            f[t]++;
        }
    }
}

int check(int x[], int pos) {
    for(int i = 1; i < pos; i++)
        if(abs(x[i] - x[pos]) == abs (i - pos) or x[i] == x[pos])
            return 0;
```

```
    return 1;
}
```

TSP

递归求解:

```
a[n][n]; //邻接矩阵，存储任意两个城市间的代价;
bestx[n]; //存储当前最小代价对应的路线;
bestc = MaxInt; //存储当前最小代价
cc = 0; //存储当前代价

void Traveling<Type>::Backtrack(int t) { //t的初值为2;
    if (t > n) {
        bestc = cc; bestx = x;
    } else {
        for (int j = t; j <= n; j++) {
            if (check(x, j, t, a, n)) {
                swap(x[t], x[j]);
                if (t < n && cc + a[x[t - 1]][x[t]] < bestc) {
                    cc = cc + a[x[t - 1]][x[t]];
                    backtrack(t + 1);
                    cc = cc - a[x[t - 1]][x[t]];
                }
                if (t == n && cc + a[x[t - 1]][x[t]] + a[x[n]][x[1]] < bestc) {
                    cc = cc + a[x[t - 1]][x[t]] + a[x[n]][x[1]];
                    backtrack(t + 1);
                    cc = cc - a[x[t - 1]][x[t]] - a[x[n]][x[1]];
                }
                swap(x[t], x[j])); //恢复现场
            }
        }
    }
}

void check (int[] x, int j, int t, int[][] a, int n) {
    if(t < 2) return 1;
    if(t < n && a[x[t-1]][x[j]] != NoEdge) return 1;
    if(t = n && a[x[t-1]][x[j]] != NoEdge && a[x[j]][x[1]] != NoEdge ) return 1; }
    return 0;
```

```
}
```

装载问题

```
r=sum(w); // 1~n个集装箱的重量和;
n; // 物品的数量;
bestx; // 最优解;
bestw; // 最优重量;
cw = 0; // 当前物品的重量;
void backtrack (int t) {// 搜索第t层结点
      if (t > n)  // 到达叶结点
      更新最优解bestx,bestw;return;
      r -= w[t];//t+1~n个集装箱的重量和
      for(int i  = 1; i ≥ 0; i--) {
          x[t] = i;
          if(cw + w[t] * i ≤ c && cw + w[t] * i + r > bestw) {
              cw += w[t] * i;
              backtrack(i + 1);
              cw -= w[t] * i ;
          }
      }
      r += w[t];//向上回溯的时候得加上
}
```

8 分支限界法

搜索方法：队列式（FIFO）搜索法；优先队列式搜索法

上界UB(v), 下界LB(v)

　　节点v上界(UB(v))：从v出发得到的所有叶子节点的效益值均**不大于**UB(v)，则UB(v)为节点v的上界；如果所有叶子节点的最大效益值等于UB(v)，则UB(v)为节点v的**上确界**；（下界与下确界同理）

- 对于求最小值的优化问题，如果LB(v) >= cBest，则节点v可以加入黑名单，不再对其搜索。UB(v)通常可以利用贪心思路或者其它方式得到一个解，令其作为UB(v)，而LB(v)通常需要经过严格的证明
- 对于求最大值的优化问题，如果UB(v) <= cBest，则节点v可以加入黑名单，不再对其搜索。LB(v)通常可以利用贪心思路或者其它方式得到一个解，令其作为LB(v)，而UB(v)通常需要经过严格的证明。

9 遗传算法

组成：

- 编码（产生初始种群）
- 适应度函数
- 遗传算子（选择、交叉、变异）
- 运行参数

遗传算法对一个个体（解）的好坏用**适应度函数值**来评价，适应度函数**设计标准**是：**适应度函数值越大，解的质量越好**。适应度函数是遗传算法进化过程的驱动力，也是进行自然选择的唯一标准，它的设计应**结合求解问题本身的要求**而定。

遗传算法使用选择运算来实现对群体中的个体进行优胜劣汰操作：适应度高的个体被遗传到下一代群体中的概率大；适应度低的个体，被遗传到下一代群体中的概率小。选择操作的任务就是按某种方法从父代群体中选取一些个体，遗传到下一代群体。SGA中选择算子采用轮盘赌选择方法。

步骤：

- 计算群体中所有个体的适应度函数值；
- 利用比例选择算子的公式，计算每个个体被选中遗传到下一代群体的概率；
- 采用模拟赌盘操作（即生成0到1之间的随机数与每个个体遗传到下一代群体的概率进行匹配）来确定各个个体是否遗传到下一代群体中。

交叉算子：

所谓交叉运算，是指对两个相互配对的染色体依据交叉概率 Pc 按某种方式相互交换其部分基因，从而形成两个新的个体。**交叉运算是遗传算法区别于其他进化算法的重要特征**，它在遗传算法中起关键作用，是产生新个体的主要方法。 SGA中交叉算子采用单点交叉算子。

遗传算法特点：

- 群体搜索，易于并行化处理；
- 自组织、自适应和自学习特征；
- 不需要求导和其它辅助知识，只需要知道适应度函数；
- 强调概率转换规则，而非确定的转换规则。

与算法收敛性有关的因素：种群规模、选择操作、交叉概率、变异概率

通常，种群太小则不能提供足够的采样点，以致算法性能很差；种群太大，尽管可以增加优化信息，阻止早熟收敛的发生，但无疑会增加计算量，造成收敛时间太长，表现为收敛速度缓慢。

Other

解空间

排列树：TSP 问题、批作业调度问题、电路板排列问题

子集树：n 皇后问题、装载问题、0-1 背包问题、最大团问题、图的 m 着色问题，高精度数问题、布线问题