邻域

```
// 01离散型一邻域
int** create_nbh_one(int *x,int x_size,int* return_size){
    int** nbh_one =  (int**)malloc(sizeof(int*) * x_size);
    for(int i = 0;i < x_size;++i){
        int bit_size = sizeof(int) * x_size;
        nbh_one[i] = (int*) malloc(bit_size);
        memcpy(nbh_one,x,bit_size);
        nbh_one[i][i] = !nbh_one[i][i];
    }
    *return_size = x_size;
    return nbh_one;
}
// 连续型一邻域,step:步长
typedef double any_type
any_type** create_nbh_one(any_type *x,int x_size,any_type step,int* return_size)
{
    any_type** nbh_one =  (any_type**)malloc(sizeof(any_type*) * x_size * 2);
    for(int i = 0;i < x_size;++i){
        int bit_size = sizeof(any_type) * x_size;
        nbh_one[i] = (any_type*) malloc(bit_size);
        memcpy(nbh_one,x,bit_size);
        nbh_one[i][2*i] +=  step;
        nbh_one[i][2*i + 1] -= step;
    }
    *return_size = x_size * 2;
    return nbh_one;
}
```

二叉树遍历

```
#define MAX_DEEP 100
struct {
    tree_node* left;
    tree_node* right;
    int weight;
} tree_node;
// 保存路径记录
tree_node* routes[MAX_DEEP];
// 目标节点深度
int deep = -1;
// 1. 权重之和为k
// 递归深度优先查找
bool find(tree_node* node,int sum,int k ,int d){
    if(node == null) return;
    sum += node->weight;
    route[d] = node;
    if(sum == k){
        deep = d;
```

```
        return 1;
    }
    return find(node.left,sum,k,d+1) || find(node.right,sum,k,d+1);
}
// 2. 权值为k的节点
bool find(tree_node* node,int k ,int d){
    if(node == null) return;
    route[d] = node;
    if(node->weight == k){
        deep = d;
        return 1;
    }
    return find(node.left,k,d+1) || find(node.right,k,d+1);
}
// 3. 树的深度
int maxDeep(tree_node* node){
    if(node == null) return 0;
    return Math.max(maxDeep(left),maxDeep(right)) + 1;
}
```

交叉算子-随机选择

```
// 交换某个位置前的序列，具体看题目要求
void crossover_random_bits(bool* lhs,bool* rhs,int geneLength) {
    if (geneLength <= 0) return;
    int flagIndex = 1.0 * rand() / RAND_MAX * (geneLength - 2) + 1; // 1 ~
geneLength - 1;
    int BITS = sizeof(bool) * flagIndex;
    bool* temp = (bool*)malloc(BITS);
    memcpy(temp, lhs, BITS);
    memcpy(lhs, rhs, BITS);
    memcpy(rhs, temp, BITS);
    free((void*)temp);
}
```

选择算子-轮盘法

```
// 给出适应值，返回一个选择的索引
int select_roulette(double* fitness, int f_size) {
    double* p = (double*) malloc(sizeof(double) * f_size);
    p[0] = fitness[0];
    for (int i = 1; i < f_size; i++)
        p[i] = p[i - 1] + fitness[i];
    // 适应值全为零
    if (p[populationSize - 1] <= 0)
        return (int)1.0 * rand()/RAND_MAX * f_size;
    double flag = 1.0 * rand()/RAND_MAX * p[f_size - 1];
    int index = 0;
    while(index < f_size)
        if (flag < p[rtn++]/p[f_size - 1]) break;
```

```
    free((void*)p);
    return index;
}
```