Data Structure and Algorithms

B.W. Sc44964

[Introduction]

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every' enterprise application uses various types of data structures in one or the other way.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Here in the club,we need to learn by ourself .Though I have found it's hard,difficult,tough.....But nothing of great significance is easy to get .All we need to do is to put all our will against here ,and it's amazing what can be achieved .

So let's start the hard way .

Please remember,

所谓无底深渊，下去也是前程万里。

B.W.

2017.4.1

[Reference]:*Data Structures and Algorithm Analysis in C*

[Website Resources]:

https://www.tutorialspoint.com/data_structures_algorithms/data_structure_overview.htm

https://www.hackerrank.com/dashboard

Alogrithm in C [part one]

基础知识

第1章 引言

对于连通性问题(Dynamic Connectivity)，书中首先提出的是快速查找算法(quick-find algorithm)。大意是把连通的id值赋值为同一个，用id值的等价来表示它的连通。

过程大致如下：

1. 初始化数组
2. 获取输入
    - 如果输入的两个值的id相等，则跳出（已连通）
    - 否则，就把所有id值为id[p]的元素变为id[q]，相同的id表示它们在同一个集合（标志连通属性）并在下一次循环中输出上一次的结果（未连通则输出p-q，连通则不输出）

则有代码：

快速查找算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, p, q, t, id[N];
  // 初始化数组
  for(i = 0; i < N; i++)
    id[i] = i;
  while(scanf(" %d %d\n", &p, &q) == 2){
    // 如果输入的两个值得id相等，则跳出（已连通）
    if(id[p] == id[q])
    {
          printf(" %d %d之前已连通\n", p, q);
          continue;
    }

    // 检查所有id值为id[p]的元素，并将其值置为id[q]
    for(t = id[p], i = 0; i < N; i++)
      if(id[i] == t)
        id[i] = id[q];
    printf(" %d %d之前未连通\n", p, q);
  }
  return 0;
}
```

优点：查找快速

缺点：慢速合并，对于每个合并操作，for循环迭代N次，N个对象，至少执行MN次，在大量的对象和输入时不可取。

但也可按照根(root)查找，根一定是指向自身的，每一次输入，都是“枝”的生长、合并，当查找的时候，找的是位于“树根”（枝的尽头）处的值，只要“树根”处的值位于一个集合中，则它们在一个“树”上，那我们就可以说，它们是连通的，这就是快速合并算法。

那么此时的数组就比第一种更加抽象了，数组的id值其实是该节点的“上一个值”，即父节点，通过数组的不停迭代上溯查找，来找到“树根”。

则有代码：

快速合并算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N];
  // 初始化数组,每一个元素都是自己的“根”
  for(i = 0; i < N; i++)
    id[i] = i;
  while(scanf(" %d %d\n", &p, &q) == 2){
    // 两个for循环都是为了找到各自的“根”。如果i和id值相等，那么
    // 自然有i == id[i]，即找到了“根”，退出循环；如果i和id值不
    // 等，则此值一定会通过“上溯”的方法找到“根”的位置，即，把其
id
    // 值作为新的查找值，直至找出其“根”。
    for(i = p; i != id[i]; i = id[i])
      ;
    for(j = q; j != id[j]; j = id[j])
      ;
    // 如果“根”相同，则说明它们在同一个集合中；否则，就将其归入
    // 另外一个集合中。
    if(i == j)
      continue;
    id[i] = j;
    printf(" %d %d\n", p, q);
  }
}
```

但是考察一下第二个算法，当进行树的合并(id[i]=j)的时候，如果把较大的树合并到较小的树上，就会造成查找的耗时。而我们可以通过增加一个判断，使较小的树总是连接到较大的树上，从而达到优化算法的目的。

这样的话，就需要一个size数组来记录每一个树的大小，在连接时进行比较，由此，将较小的树连接到较大的树上。

则有代码：

加权快速合并算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N], sz[N];
  for(i = 0; i < N; i++){
    id[i] = i;
    sz[i] = 1; // 每一个独立单元都是一个size为1的树
  }
  while(scanf(" %d %d\n", &p, &q) == 2){
    for(i = p; i != id[i]; i = id[i])
      ;
    for(j = q; j != id[j]; j = id[j])
      ;
    if(i == j)
      continue;
    // 加权合并
    if(sz[i] < sz[j]){
      id[i] = j;
      sz[j] += sz[i];
    }else{
      id[j] = i;
      sz[i] += sz[j];
    }
    printf(" %d %d\n", p, q);
  }
}
```

优点：树中每个节点到根节点的距离变小，因而查找效率更高，非常高。

但对于这个算法而言，我们可以把一个树进行路径压缩，使树“平扁化”。

等分路径压缩

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N], sz[N];
  for(i = 0; i < N; i++){
    id[i] = i;
    sz[i] = 1; // 每一个独立单元都是一个size为1的树
  }
  while(scanf(" %d %d\n", &p, &q) == 2){
    for(i = p; i != id[i]; i = id[i])
      id[i] = id[id[i]];
    for(j = q; j != id[j]; j = id[j])
      id[j] = id[id[j]];
    if(i == j)
      continue;
    // 加权合并
    if(sz[i] < sz[j]){
      id[i] = j;
      sz[j] += sz[i];
    }else{
      id[j] = i;
      sz[i] += sz[j];
    }
    printf(" %d %d\n", p, q);
  }
}
```

第一章通过对连通性问题求解的逐步求精给我们展示了一个算法从雏形到完善的整个过程，这一章讲算法的目的并不在于解决这一问题，而在于展现出设计算法的基础步骤：

1. 确定完整、明确的问题描述，包括确定问题固有的基本抽象操作。
2. 仔细设计一个简单算法的简明实现。
3. 通过逐步求精的过程开发改进后的实现，经过实验分析、数学分析或两者共同验证改进思想的效率。
4. 找出数据结构或者算法操作的高级抽象表示，能够使改进版本的设计性能高效。
5. 可能时尽量保证最坏情况下的性能，但实际数据可用时接受好的性能。

而对于第一章的学习，我们可以知道数组的抽象形式。一个整型数组不仅仅只是内存中占有等长字节的连续的一组值，它可通过位-值得关系来进行排序（桶排序），也可以抽象成集合、树（如上），甚至还可以对树进行加权、压缩！

And.....

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

所以我们下面再开始分析算法吧。。。。（前言还有一些超级变态的课后题，一起来做做吧。。。）

第2章 算法分析的原理