**课程设计报告**

20307110315 周训哲 计算机科学与技术

**一、需求分析**

**1寻找公共祖先**：根据层次遍历进行编号,使用Tarjan离线查找算法，输入两个结点的编号就可以返回已经变异过的两个结点的最近公共祖先。

**2沙箱模拟：**每过一个小时对每个人进行状态的更新，每个小时所有病毒传播一次；每天病毒进行一次变异操作，每天对当前每个病毒感染状态进行输出。

2.1病毒变异：每天所有可变异的病毒会进行变异操作

判断是否变异、设置变异时间的设置、变异与疫苗研究。

2.2病毒传染：每个已经变异过的病毒每小时传染一次，以传染率为基准， 每次传染人数为以其为期望值的正态分布统计值，方差为0.5。

2.3病毒爆发：当感染时间超过病毒的潜伏期后，病毒会爆发性传染，此时 传染率为原来10倍。

2.4病毒隔离：病毒爆发一段时间后，会将病人进行隔离，隔离后病毒将无 法传染，其传染率为0，沙箱中的爆发到隔离的时间为0~MAXDAY的一个随机 数。

2.5感染人类状态转移：每小时对于每个感染病毒的人类会进行状态的转移， 分别为治愈，继续治疗，死亡。

（bonus）2.6疫苗研究：如果某种病毒的疫苗成功研制，则该病毒的传染率 为0，且所有病人康复。

（bonus）2.7沙箱交换：每天随机交换不同沙箱当中的人员，并继续对不同 沙箱进行模拟。

**3时针转换及输出**：每过一个小时进行一次时针转换，并且时针每走过24小时，天数增加，将当天感染情况输出。

**二、概要设计**

**1寻找公共祖先**：使用Tarjan离线查找算法,在自根结点向下遍历结点时就已经将结点的信息存储到并查集中，只需利用并查集存储结果即可。最终实现，输入两个病毒结点的代号输出其最近的已经变异的祖先结点的代号。

**1.1使用到的数据结构：**

并查集：class Union {//并查集的数据结构

private:

VirusTree \*virus;//并查集所包含的树

vector<int>pre;//前驱结点

vector<int>height;//height[x]表示以x为根的树的高度

vector<int>vis;//判断结点是否被遍历

public:

Union();//构造函数

~Union();//析构函数

void Initialize();//并查集初始化

void SetTree(VirusTree \*Tree);//设置并查集当中的树

int UnionFind(int x);//查找根节点

bool UnionisConnected(int x, int y);//判断两个结点是否在一个集

bool UnionJoin(int x, int y);//将两个元素置于同一个集

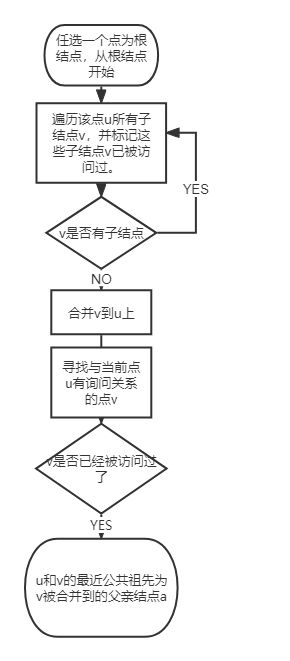
void UnionVis(int x);//对遍历过的结点标记

bool UnionisVis(int x);//判断是否标记

VirusTree\* GetVirus();//返回病毒树

};

**1.2主要实现算法设计说明：**



**2沙箱模拟**

**2.1使用到的数据结构：**

病毒树：class VirusTree;

class Union {//并查集的数据结构

private:

VirusTree \*virus;//并查集所包含的树

vector<int>pre;//前驱结点

vector<int>height;//height[x]表示以x为根的树的高度

vector<int>vis;//判断结点是否被遍历

public:

Union();//构造函数

~Union();//析构函数

void Initialize();//并查集初始化

void SetTree(VirusTree \*Tree);//设置并查集当中的树

int UnionFind(int x);//查找根节点

bool UnionisConnected(int x, int y);//判断两个结点是否在一个集

bool UnionJoin(int x, int y);//将两个元素置于同一个集

void UnionVis(int x);//对遍历过的结点标记

bool UnionisVis(int x);//判断是否标记

VirusTree\* GetVirus();//返回病毒树

};

病毒结点：struct VirusNode {//树节点结构的定义

VirusNode();//构造函数

~VirusNode();//析构函数

bool ifLeaf();//判断是否是叶子结点

int Incubation;//潜伏期:为整数,以天为单位

double Infection;//传染性:为实数,每小时感染多少人

//潜伏期的传染能力为爆发期的1/10

double DeathRate[3];//致死率:DeathRate[0]表示康复率

//DeathRate[1]表示持续治疗概率

//DeathRate[2]表示死亡率

int VariationTime;//变异所需的时间

int order;//结点序号

int stage;//病毒状态,0:潜伏期,1:爆发期,2:隔离救治期

int timecount;//病毒内置时钟

int begintime;//病毒初次感染的时间

bool ifInfect;//标记是否感染过

int VaccineTime;//疫苗完成制作所需时间

int VaccineBeginTime;//疫苗开始制作的时间

bool ifVaccineSucc;//判断疫苗是否研制成功

vector<Human\*> Infected;//正在感染的人

vector<Human\*> Recover;//已经康复的人

vector<Human\*> Death;//已经死亡的人

VirusNode\* subVirus[N];//子结点

VirusNode\* PreVirus;//父亲结点

};

人类结构：struct Human {//人类结构

VirusTree\* virus;//人类结构内置病毒树

vector<VirusNode\*>InfectedVirus;//存储感染过的病毒

vector<int>mark;//标记是否被该病毒感染过(全体病毒)

Human();//构造函数

~Human();//析构函数

void Initialize(VirusTree\* Tree);//人类结构初始化

};

沙箱结构：class SandBox {//沙箱结构:每一种病毒存储了感染该病毒的人数,每一个人存储了其感染的病毒类型

private:

VirusTree\* virus;//该沙箱中的病毒

vector<Human\*> HumanSet;//沙箱人类

int Population;//人口数

int IsolationTime;//爆发到隔离所需时间

public:

SandBox();//构造函数

~SandBox();//析构函数

int GetPopulation();//输出此时人口总数

void SetTree(VirusTree\* Tree);//初始化沙箱病毒树

vector<Human\*> GetHumanSet();//初始化人类集

void Infect(VirusNode\* current);//进行感染过程

void SwitchSituation();//沙盒时间增加

void Clkdiv();//计数器并且输出

void SetHuman(VirusTree\* Tree);//初始化HumanSet

void SwicthState(int humanorder);//病人的状态转换

VirusNode\* Variation(VirusNode\* current);//病毒变异

void Break(VirusNode\* current);//病毒爆发

void Isolate(VirusNode\* current);//病毒隔离治疗

void Vaccine(VirusNode\* current);//生产疫苗

void Recover(VirusNode\* current, Human\* hm);//恢复

void Kill(VirusNode\* current, Human\* hm);//死亡

void UpdateMark();//更新每个人的病毒感染记号

bool ifSucc();//判断是否成功抵御病毒

int HourCnt, DayCnt;//时间变量

VirusTree\* GetVirus();//返回病毒树

};

**2.2病毒变异：**生成一棵度数为N的树，对每个可以变异的结点进行遍历，如果该节点是叶子结点，则生成N个子结点，对其所有子结点进行判断，看是否可以变异，若可以变异，则随机选择一个结点作为变异结点进行变异（用rand（）函数实现），若该节点不是叶子结点，则在其子结点中随机找一个可以变异的结点进行变异（用rand（）函数实现）。

2.2.1判断是否能够变异：如果疫苗研制时间小于变异所需时间，则无法变异；如果病毒感染时间大于其中几个子结点变异所需时间，则在这几个子结点中随机抽取一个结点进行变异，若没有可变异的，则不变异。若无法变异返回NULL。

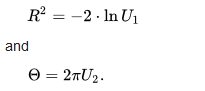
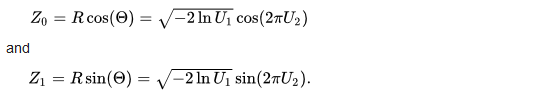
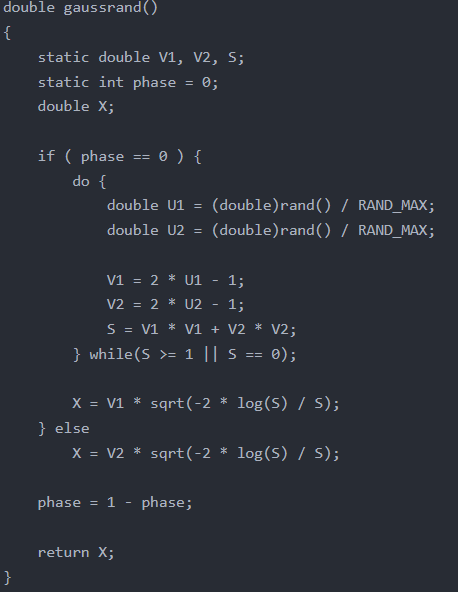
2.2.2变异时间的设置：每个结点的变异时间初始值设置为0~MAXDAY的一个随机数（用rand（）函数实现），MAXDAY根据输入而定。

2.2.3变异与疫苗研究：如果可以变异且疫苗研制成功，则随机延长其变异时间为原变异时间~2\*原变异时间（用rand（）函数实现）。

**2.3病毒传染：**每个已经变异过的病毒每小时传染一次，以传染率为基准，每次传染人数为以其为期望值的正态分布统计值，方差为0.5。

每次传染的人数为正态分布统计值，即随机抽取其数量的人数感染病毒，若存活人数大于于感染率统计值，则相应数量的人感染（可重复），若存活人数小于等于感染率统计值，则全部人都感染。

其中高斯分布随机数的实现如下：



利用[Box–Muller transform](https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform)算法即可将均匀分布的随机数（用rand（）函数实现）转换为高斯分布的钟形曲线的随机数，由上实现的随机数均值为0，方差为1，若指定期望为E，标准差为V，则只需增加：X = X \* V + E;(由Marsaglia和Bray在1964年提出)

感染之后将感染的人类加入到该病毒感染的人类集中。

**2.4病毒爆发：**当感染时间超过病毒的潜伏期后，病毒会爆发性传染，此时传染率为原来10倍，只需修改病毒结构的参数即可。将病毒的状态从潜伏期状态设置为爆发期状态。

**2.5病毒隔离：**病毒爆发一段时间后，会将病人进行隔离，隔离后病毒将无法传染，修改其传染率为0，沙箱中的爆发到隔离的时间为0~MAXDAY的一个随机数（用rand（）函数实现），沙箱初始生成的构造函数中就已经设定。将病毒的状态从爆发期状态设置为隔离期状态。

**2.6感染人类状态转移：**每小时对于每个感染病毒的人类会进行状态的转移，分别为治愈，继续治疗，死亡。

初始设置病毒节点的致死率为三元组，其治愈率、继续治疗的概率、死亡概率之和为1（其概率随机生成用rand（）函数实现），

治愈：患者治愈后不会再感染该种病毒以及其所有祖先。将病毒结点从康复的人类感染的病毒集中删除，将该人类从病毒感染的人类集中删除；并且将该人类放入该病毒的治愈的人类集中。

继续治疗：维持感染状态继续接受治疗，一小时后进行下一次状态转移。

死亡：人类死亡，从人类集中删除该人类，并且从该病毒感染的人类集中删除该人类，将该人类加入到该病毒杀死的人类集中。

三者状态的转换为体现随机性，用rand（）函数随机生成0~1的概率数，落在哪个区间，人类便进行哪个类型的转换。

**（bonus）2.7疫苗研究**：如果某种病毒的疫苗成功研制，即研制时间加开始研制时间小于变异时间加开始传染时间，则将该病毒的传染率设置为0。并且调整该病毒的变异时间，且所有病人康复（康复操作见2.6的治愈环节）。

**（bonus）2.8沙箱交换：**每天随机交换不同沙箱当中的人员，并继续对不同沙箱进行模拟。合并病毒树与合并病毒库:

1.本项目中,沙箱操作/人类结构的操作/并查集的操作全部是基于树的结构进行，如果人员流动,则需要引入树的添加与删除操作(基于判断病毒宿主是否全部交换的基础上)，且两棵不同的树的合并只是基于部分结点,因此生成的树有交叉或无交叉,需要判断;

2.由于算法设计时不同沙箱病毒树的随机性,所以合并病毒树与病毒库的工作是基于大量查找的基础上,以保证病毒结点的不重合;

3.对于存活人数不同的沙箱,交换的人数也要有设定

4.交换后的病毒树由于有查找根结点的操作,可以在原有病毒树上实现

5.需对每个人类及病毒进行遍历,以判断所有结点情况

**3时针转换及输出**

每过一个小时进行一次时针转换，并且时针每走过24小时，天数增加，将当天感染情况输出。输出序列：变异的病毒编号：感染情况，感染状态，治愈人数，死亡人数，如果疫苗未研制成功，需延后的时间。

最后如果病毒无法传染且有人类存活,则人类胜利；如果人类全部死亡，则无人生还。

**三、 详细设计**

#include"Human.h"

#include"Node.h"

#include"Tree.h"

#include"Union.h"

#include"SandBox.h"

using namespace std;

#define N 20//结点度数

int MAXDAY;//潜伏期,变异时间,疫苗研制时间所需最大整数

int POPULATION;//沙箱人口总数

主函数：

int main() {

srand((unsigned)time(NULL));//蓝星病毒沙箱模拟

cout << "Enter MAXDAY: " << endl;

cin >> MAXDAY;

cout << "Enter POPULATION: " << endl;

cin >> POPULATION;

VirusTree\* Tree = new VirusTree;

Tree->SaveEveryNode();

Union\* Un = new Union;

SandBox\* Sb = new SandBox;

Sb->SetTree(Tree);

Sb->SetHuman(Tree);

int date = 1;

while (Sb->GetPopulation() > 0 || Sb->HourCnt % 24 != 0) {

//时针转动的条件:天数没有增加或人口数大于0

Sb->Clkdiv();

Sb->SwitchSituation();

if (Sb->ifSucc() && Sb->HourCnt % 24 == 0)

break;

}

if (Sb->ifSucc())//如果病毒无法传染且有人类存活,则人类胜利

cout << endl << "Humankind wins!" << endl;

else cout << endl << "No man suvives!" << endl;//如果人类全部死亡，则无人生还

int a, b;//寻找两个结点最近的存在的公共祖先

Un->SetTree(Sb->GetVirus());

Un->Initialize();

cout << "Enter 2 numbers of viruses: " << endl;

cin >> a >> b;

if (Un->GetVirus()->NodeSet.size() == 1 &&

(a <= Un->GetVirus()->NodeSet.size() && b <= Un->GetVirus()->NodeSet.size()))

{//树中只有一个结点

cout << "Tree has only one node!" << endl;

}

else if (a > Un->GetVirus()->NodeSet.size() || b > Un->GetVirus()->NodeSet.size()) {

//结点不存在

cout << "Order NOT exist!" << endl;

}

else {

VirusNode\* tmp = new VirusNode;

stack<VirusNode\*>stmp;

for (int i = 0; i < Un->GetVirus()->currentNodeSet.size(); i++)

{

tmp = Un->GetVirus()->GetNearestAncestor

(Un, Un->GetVirus()->NodeSet[0], Un->GetVirus()->NodeSet[a], Un->GetVirus()->NodeSet[b]);

while (tmp != NULL) {

//如果最近祖先在存在的病毒集中，则输出，否则继续寻找根结点

tmp = tmp->PreVirus;

if (tmp == Un->GetVirus()->currentNodeSet[i])

stmp.push(tmp);

}

}//输出最后的结点

cout << "Nearest ancestor is: Node " << stmp.top()->order;

}

}

1、给定两个病毒的编号，找出一个变异体，这个变异体是这两个病毒在分化之

前的最近公共祖先。

VirusNode\* VirusTree::GetNearestAncestor(Union\* Un, VirusNode\* root, VirusNode\* virus1, VirusNode\* virus2) {

//Tarjan离线算法,由于从上到下遍历时就已经可以判断最近祖先结点了,单次遍历即可,复杂度为O(n+q)

if (root != NULL) {

for (int i = 0; i < N; i++) {

if (root->subVirus[i] != NULL) {

GetNearestAncestor(Un, root->subVirus[i], virus1, virus2);//深度优先遍历节点

Un->UnionJoin(root->order, root->subVirus[i]->order);//将遍历到的结点存入同一个并查集

Un->UnionVis(root->subVirus[i]->order);//标记节点已经遍历

}

}

if (virus1 == root && Un->UnionisVis(virus2->order)) {

return NodeSet[Un->UnionFind(virus2->order)];//如果遍历到virus1,virus2标记过,则返回2的根结点

}

if (virus2 == root && Un->UnionisVis(virus1->order)) {

return NodeSet[Un->UnionFind(virus1->order)];//如上同理

}

}

}

Union::Union() {

virus = NULL;

}

Union::~Union() {

pre.clear();

height.clear();

vis.clear();

}

void Union::Initialize() {//初始化并查集

for (int i = 0; i < virus->NodeSet.size(); i++) {

pre.push\_back(i);

height.push\_back(1);

vis.push\_back(0);

}

}

void Union::SetTree(VirusTree\* Tree) {

virus = Tree;

}

int Union::UnionFind(int x) {//查找结点的根结点

if (pre[x] == x) return x;

return pre[x] = UnionFind(pre[x]);

}

bool Union::UnionisConnected(int x, int y) {//判断两个结点是否在一个并查集

return UnionFind(x) == UnionFind(y);

}

bool Union::UnionJoin(int x, int y) {//将x合并到y的并查集

x = UnionFind(x);

y = UnionFind(y);

if (x == y) return false;

else {

if (height[x] == height[y]) height[y]++;

pre[x] = y;

}

return true;

}

void Union::UnionVis(int x) {

vis[x] = 1;

}

bool Union::UnionisVis(int x) {

if (vis[x] == 1) return true;

else return false;

}

VirusTree\* Union::GetVirus() {

return virus;

}

2、若科学家沿某一路径研制疫苗时间比病毒沿某一路径变异所需要的时间短，

则该病毒不会再进行变异。

VirusNode\* SandBox::Variation(VirusNode\* current) {//变异过程是先生成树,然后再变异,可以动态节省空间

if (current->VariationTime <= current->VaccineTime && current->ifLeaf()) {

//如果该节点是叶子结点,则生成N个子结点,并从中查找可变异结点

int k = 0;

vector<VirusNode\*>vari;//定义变异概率

for (int i = 0; i < N; i++) {

VirusNode\* subnode = new VirusNode;//生成变异结点(创建树的过程)

current->subVirus[i] = subnode;

subnode->PreVirus = current;

if ((int)(current->timecount / 24)

>= current->subVirus[i]->VariationTime) {

//变异:如果时针大于变异所需时间,则向某个节点变异

k++;

vari.push\_back(current->subVirus[i]);//将可变异的结点暂存

}

}

if (k > 0) {

int random = rand() % k;//在可变异的子结点中随机取一个

current = vari[random];

return current;

}

}

else if (current->VariationTime <= current->VaccineTime && current->ifLeaf()) {

//如果该节点不是叶子结点,则从其子结点中查找可变异结点

int k = 0;

vector<VirusNode\*>vari;//定义变异概率

for (int i = 0; i < N; i++) {

if ((int)(current->timecount / 24)

>= current->subVirus[i]->VariationTime) {

//变异:如果时针大于变异所需时间,则向某个节点变异

k++;

vari.push\_back(current->subVirus[i]);

}

}

if (k > 0) {

int random = rand() % k;//在可变异的子结点中随机取一个

current = vari[random];

return current;

}

}

return NULL;

}

3、假定所有变异和疫苗研究可以并行进行，即每种病毒在诞生后，经过相应时

间后会变异出其所有孩子结点的病毒(病毒会分头行动，并行变异)，假定研制疫

苗不一定需要这种病毒已经出现，请编写程序计算每种变异体的疫苗是否能在该变异出现前研制成功，若不能，会落后多长时间。

bool SandBox::ifSucc(){//通过病毒能否继续传染判断人类是否胜利

for (int i = 0; i < virus->currentNodeSet.size(); i++) {

if (virus->currentNodeSet[i]->Infection != 0)

return false;

}

if (HumanSet.size() > 0)

return true;

else

return false;

}

void SandBox::Vaccine(VirusNode\* current) {//进行疫苗的研制并设置病毒疫苗研制状态

if ((current->timecount + current->begintime) /24

>= (current->VaccineTime + current->VaccineBeginTime / 24)) {

current->ifVaccineSucc = true;//疫苗研制完成

current->Infection = 0;//将病毒传染率设置为0,即不能继续传染

if (current->VariationTime != 0)

current->VariationTime = rand() % current->VariationTime + current->VariationTime + 1;

//疫苗研制成功后变异时间将随机增加为VariationTime~2\*VariationTime区间

for (int i = 0; i < current->Infected.size(); i++) {//每一个病人康复

Recover(current, current->Infected[i]);

}

}

}

4、对病毒在人群中的传播和变异进行一次沙箱模拟，以天为单位输出，以小时

为每一次迭代的粒度。规则如下：

(1)、潜伏期为整数，以天为单位。

(2)、传染能力为一个实数，即每个感染者每小时会传染多少易感者，为简化

问题，假定感染者之间互相传播也计算在内。

(3)、潜伏期的传染能力为爆发期的十分之一，爆发之后患者将被隔离，隔离

后将无法传染。爆发到隔离的时间由输入决定，为一个定值。

(4)、致死率为一个三元组，每个感染的人每小时状态会发生一次转移，分别

是康复、持续以及死亡，转移到持续状态的即为继续治疗状态，一小时后进行

下一次状态转移。

(5)、在某种病毒感染到达相应天数后，则相应感染人群携带的病毒即会有 1/k

的概率变异为对应的病毒，该天数包括潜伏期。例: 假若有一个病毒有三个变

异体，且每一个病毒需要不同的变异时长，若其中两个变异体变异所需时间较

短，另一个较长。在到达较短时间时，

"对应的病毒"指这两个变异体中的一个，

每一个概率为 1/2；若三个变异体的变异时间都到达，"对应的病毒"指三个中

的其中一个，每一个概率为 1/3。K 会根据到达变异时间的变异体数决定。

(6)、感染孩子结点的病毒并康复后不会再次感染这种病毒及其所有祖先，但

是有可能感染除此以外的变异体。

(7)、bonus（选做）：加入疫苗的研制，当某种疫苗研制成功后，将不会再有人

感染相应病毒。

(8)、bonus（选做）：将沙箱进行分割，即存在多个沙箱，沙箱之间会有人员流

动，如每天随机交换 10%的人员，不同沙箱从爆发到隔离的时间不同。

请自行决定输入方式，以天为粒度输出每一天每种病毒的感染、康复以及死亡的

人数，感染可分为潜伏期、爆发期以及隔离救治期。

Human::~Human() {

InfectedVirus.clear();

}

Human::Human() {

virus = NULL;

}

void Human::Initialize(VirusTree\* Tree) {//内置树初始化

virus = Tree;

}

VirusNode::VirusNode() {//每个结点进行初始化

Incubation = rand() % MAXDAY;//潜伏期的设定范围为0~MAXDAY的随机数

Infection = rand() / (double)(RAND\_MAX) \* POPULATION;

//传染率的设定范围为0~POPULATION的double类型的随机数

double a = rand()%100, b = rand()%100, c = rand()%100;

double total = a + b + c;

DeathRate[0] = a / total; DeathRate[1] = b / total; DeathRate[2] = c / total;

//致死率的设定随机生成,三者概率总和加起来为1

//DeathRate[0]表示康复率

//DeathRate[2]表示持续治疗概率

//DeathRate[3]表示死亡率

VariationTime = rand() % MAXDAY;//变异所需的时间范围为0~MAXDAY的随机数

order = 0;

Infected.clear(); Recover.clear(); Death.clear();

stage = 0;

timecount = 0;

begintime = 0;

VaccineBeginTime = 0;

VaccineTime = rand() % MAXDAY;//疫苗研制所需时间范围为0~MAXDAY的随机数

ifInfect = false;

ifVaccineSucc = false;

for (int i = 0; i < N; i++) {

subVirus[i] = NULL;

}

PreVirus = NULL;

}

VirusNode::~VirusNode() {

Infected.clear();

delete[] subVirus;

delete PreVirus;

}

bool VirusNode::ifLeaf() {//如果所有子结点为NULL则该节点为叶子结点

for (int i = 0; i < N; i++) {

if (subVirus[i] != NULL) {

return false;

}

}

return true;

}

double GaussRand(int Infection) {//高斯函数,得到正态分布的随机数

static double V1, V2, S;

static int phase = 0;

double x;

if (phase == 0) {

do {

double U1 = (double)rand() / RAND\_MAX;

double U2 = (double)rand() / RAND\_MAX;

V1 = 2 \* U1 - 1;

V2 = 2 \* U2 - 1;

S = V1 \* V1 + V2 \* V2;

} while (S >= 1 || S == 0);

x = V1 \* sqrt(-2 \* log(S) / S);

}

else

x = V2 \* sqrt(-2 \* log(S) / S);

phase = 1 - phase;

return (x \* 0.5 + Infection);//期望为感染率，标准差为100

}

SandBox::SandBox() {

virus = new VirusTree;

HumanSet.clear();

Population = POPULATION;//定义人数

IsolationTime = rand() % MAXDAY;//爆发到隔离所需时间随机生成

HourCnt = 0; DayCnt = 0;//沙箱内置时钟

}

SandBox::~SandBox() {

virus->~VirusTree();

HumanSet.clear();

}

int SandBox::GetPopulation() {

Population = HumanSet.size();

return Population;

}

void SandBox::SetTree(VirusTree\* Tree) {

virus = Tree;

}

vector<Human\*> SandBox::GetHumanSet() {

return HumanSet;

}bool SandBox::ifSucc(){//通过病毒能否继续传染判断人类是否胜利

for (int i = 0; i < virus->currentNodeSet.size(); i++) {

if (virus->currentNodeSet[i]->Infection != 0)

return false;

}

if (HumanSet.size() > 0)

return true;

else

return false;

}

void SandBox::Vaccine(VirusNode\* current) {//进行疫苗的研制并设置病毒疫苗研制状态

if ((current->timecount + current->begintime) /24

>= (current->VaccineTime + current->VaccineBeginTime / 24)) {

current->ifVaccineSucc = true;//疫苗研制完成

current->Infection = 0;//将病毒传染率设置为0,即不能继续传染

if (current->VariationTime != 0)

current->VariationTime = rand() % current->VariationTime + current->VariationTime + 1;

//疫苗研制成功后变异时间将随机增加为VariationTime~2\*VariationTime区间

for (int i = 0; i < current->Infected.size(); i++) {//每一个病人康复

Recover(current, current->Infected[i]);

}

}

}

void SandBox::Infect(VirusNode\* current) {//进行病毒的传播过程

if (current->ifInfect == false) {

current->ifInfect = true;//设置病毒传染状态

current->begintime = HourCnt;//初始化病毒内置时钟

current->VaccineBeginTime = rand() % current->begintime;

//研制疫苗假定在刚传染或之前就开始进行

}

if (current->ifVaccineSucc == false) {

int intInf = (int)GaussRand(current->Infection);

//设置感染人数为围绕期望值(传染率)正太分布的数值

if(intInf<=HumanSet.size()){//对人类进行随机传染(可以重复传染)

for (int i = 0; i < intInf; i++) {

int random = rand() % HumanSet.size();//设置随机数,对不超过传染率统计值的人数进行传染

if (HumanSet[random]->mark[current->order] != 1) {//判断是否已经感染过该病毒

HumanSet[random]->InfectedVirus.push\_back(current);

HumanSet[random]->mark[current->order] = 1;

current->Infected.push\_back(HumanSet[random]);

}

}

}

else//如果传染率高于存活的人数,则对每一个存活的人进行传染

for (int i = 0; i < HumanSet.size(); i++) {

int random = rand() % HumanSet.size();

if (HumanSet[random]->mark[current->order] != 1) {

HumanSet[random]->InfectedVirus.push\_back(current);

HumanSet[random]->mark[current->order] = 1;

current->Infected.push\_back(HumanSet[random]);

}

}

}

}

void SandBox::SetHuman(VirusTree\* Tree) {//设置总的人类集合

for (int i = 0; i < Population; i++) {

Human\* hm = new Human;

hm->Initialize(Tree);//对每一个人类进行内置树的初始化

HumanSet.push\_back(hm);

}

}

void SandBox::Clkdiv() {//时针跳动,如果天数改变则输出每天的沙箱模拟情况

HourCnt++;//时针开始跳动

if (HourCnt % 24 == 0) {

DayCnt++;//如果时针转过24小时,则天数进一位

cout << "Day" << DayCnt << ": " << endl;

for (int i = 0; i < virus->currentNodeSet.size(); i++) {

cout << "virus(exist)" << i << ": " <<

"Infected: " << virus->currentNodeSet[i]->Infected.size()

+ virus->currentNodeSet[i]->Recover.size()

+ virus->currentNodeSet[i]->Death.size() <<//感染人数

" Stage: ";

switch (virus->currentNodeSet[i]->stage)

{

case 0:cout << "INCUBATE"; break;

case 1:cout << "BREAK "; break;

case 2:cout << "ISOLATE "; break;

}//感染状态

cout<< " Recover: " << virus->currentNodeSet[i]->Recover.size() <<//治愈人数

" Death: " << virus->currentNodeSet[i]->Death.size() << //死亡人数

" LagTime: ";//落后时间

if ((virus->currentNodeSet[i]->timecount + virus->currentNodeSet[i]->begintime)/24

< (virus->currentNodeSet[i]->VaccineTime + virus->currentNodeSet[i]->VaccineBeginTime / 24))

cout << (virus->currentNodeSet[i]->VaccineTime + virus->currentNodeSet[i]->VaccineBeginTime / 24)

- (virus->currentNodeSet[i]->timecount + virus->currentNodeSet[i]->begintime)/24;

else cout << "-";

cout << endl;

virus->currentNodeSet[i]->Death.clear();

virus->currentNodeSet[i]->Recover.clear();

//将每天的死亡人数与治愈人数清空,以保证当天输出的只是当天的情况

}

}

}

void SandBox::Recover(VirusNode\* current, Human\* hm) {//对人类进行治愈操作

hm->mark[current->order] = 1;//将该节点标记

current->Recover.push\_back(hm);//将恢复的人类存入

if (current->Infected.size() > 1) {//将该人类从病毒结点中感染的人类集删除

for (int j = 0; j < current->Infected.size(); j++) {

if (hm == current->Infected[j]) {

current->Infected.erase

(current->Infected.begin() + j, current->Infected.begin() + j + 1);

}

}

}

else

current->Infected.clear();

VirusNode\* temp = current;

while (temp->PreVirus != NULL) {

temp = temp->PreVirus;

hm->mark[temp->order] = 1;

}//标记该结点与祖父结点均不会被再次感染

if (hm->InfectedVirus.size() > 1) {//将病毒从该人类感染的病毒集删除

for (int j = 0; j < hm->InfectedVirus.size(); j++) {

if (current == hm->InfectedVirus[j]) {

hm->InfectedVirus.erase

(hm->InfectedVirus.begin() + j, hm->InfectedVirus.begin() + j + 1);

}

}

}

else

hm->InfectedVirus.clear();

}

void SandBox::Kill(VirusNode\* current, Human\* hm) {//进行死亡操作

current->Death.push\_back(hm);//将该人类存入死亡人类集合

if (current->Infected.size() > 1) {//将该人类从病毒的感染人类集合进行删除

for (int j = 0; j < current->Infected.size(); j++) {

if (hm == current->Infected[j]) {

current->Infected.erase

(current->Infected.begin() + j, current->Infected.begin() + j + 1);

}

}

}

else

current->Infected.clear();

if (HumanSet.size() > 1) {//将该人类从沙箱的人类集中进行删除

for (int i = 0; i < HumanSet.size(); i++) {

if (hm == HumanSet[i]) {

HumanSet.erase(HumanSet.begin() + i, HumanSet.begin() + i + 1);

}

}

}

else

HumanSet.clear();

}

void SandBox::SwicthState(int humanorder) {//状态转移:个体生存状态转移

Human\* person = HumanSet[humanorder];

for (int i = 0; i < person->InfectedVirus.size(); i++) {

VirusNode\* curvirus = person->InfectedVirus[i];

double rec, keep, die, state;

rec = curvirus->DeathRate[0];

keep = curvirus->DeathRate[1];

die = curvirus->DeathRate[2];

state = rand() / (double)(RAND\_MAX);//设置0~1的随机数

if (state >= 0 && state <= rec) {//如果随机数落在恢复区间,进行恢复操作

Recover(curvirus, person);

}

else if (state <= rec + die) {//如果随机数落在死亡区间,进行死亡操作

Kill(curvirus, person);

}

//否则处于持续治疗状态

}

}

VirusNode\* SandBox::Variation(VirusNode\* current) {//变异过程是先生成树,然后再变异,可以动态节省空间

if (current->VariationTime <= current->VaccineTime && current->ifLeaf()) {

//如果该节点是叶子结点,则生成N个子结点,并从中查找可变异结点

int k = 0;

vector<VirusNode\*>vari;//定义变异概率

for (int i = 0; i < N; i++) {

VirusNode\* subnode = new VirusNode;//生成变异结点(创建树的过程)

current->subVirus[i] = subnode;

subnode->PreVirus = current;

if ((int)(current->timecount / 24)

>= current->subVirus[i]->VariationTime) {

//变异:如果时针大于变异所需时间,则向某个节点变异

k++;

vari.push\_back(current->subVirus[i]);//将可变异的结点暂存

}

}

if (k > 0) {

int random = rand() % k;//在可变异的子结点中随机取一个

current = vari[random];

return current;

}

}

else if (current->VariationTime <= current->VaccineTime && current->ifLeaf()) {

//如果该节点不是叶子结点,则从其子结点中查找可变异结点

int k = 0;

vector<VirusNode\*>vari;//定义变异概率

for (int i = 0; i < N; i++) {

if ((int)(current->timecount / 24)

>= current->subVirus[i]->VariationTime) {

//变异:如果时针大于变异所需时间,则向某个节点变异

k++;

vari.push\_back(current->subVirus[i]);

}

}

if (k > 0) {

int random = rand() % k;//在可变异的子结点中随机取一个

current = vari[random];

return current;

}

}

return NULL;

}

void SandBox::Break(VirusNode\* current) {//病毒传染爆发

if ((int)(current->timecount / 24) >= current->Incubation && current->stage == 0) {

current->stage = 1;//将病毒状态设置为爆发状态

current->Infection \*= 10;//传染率设置为原来10倍

}

}

void SandBox::Isolate(VirusNode\* current) {

if ((int)(current->timecount / 24 - current->Incubation) >= IsolationTime && current->stage == 1) {

//判断爆发:将爆发时间与潜伏期对比判断病毒是否需要隔离

current->Infection = 0;//隔离之后将无法继续传染

current->stage = 2;//将病毒状态设置为隔离状态

}

}

void SandBox::UpdateMark() {//更新每一个人类的标记

int tmp;

for (int i = 0; i < HumanSet.size(); i++) {

tmp = HumanSet[i]->mark.size();

for (int j = 0; j < (virus->NodeSet.size() - tmp); j++)

HumanSet[i]->mark.push\_back(0);

//如果有新增结点,则在后方加入对应标志,并设置为未感染

}

}

void SandBox::SwitchSituation() {

//变异等活动由于是并行的,所以应遍历整颗树,对每个结点都进行状态变换操作

for (int i = 0; i < virus->currentNodeSet.size(); i++) {

if (virus->currentNodeSet[i]->ifVaccineSucc == false) {

VirusNode\* current = virus->currentNodeSet[i];//对节点进行遍历

if (current != NULL) {

if (current->ifInfect) {

current->timecount++;//如果结点感染过,则其内置时针转动

}

if (current->VariationTime <= current->timecount / 24

&& current->timecount % 24 == 0)

//感染天数大于变异所需天数,则进行变异操作

{

VirusNode\* tmp = Variation(current);

if (tmp != NULL) {

virus->currentNodeSet.push\_back(tmp);

//病毒变异并存入病毒库中

}

}

virus->SaveEveryNode();//更新病毒结点库

UpdateMark();//对每个人类将病毒标记更新

Infect(current);//进行感染操作

for (int i = 0; i < HumanSet.size(); i++) {

SwicthState(i);//对每一个人进行状态转移

}

Vaccine(current);//生产疫苗

Break(current);//病毒爆发

Isolate(current);//隔离治疗

}

}

}

}

VirusTree\* SandBox::GetVirus() {

return virus;

}

VirusTree::VirusTree() {

root = new VirusNode;

currentNodeSet.push\_back(root);

}

VirusTree::~VirusTree() {

DestroyTree(root);

}

void VirusTree::SaveEveryNode() {//层次遍历存储树的结点(树的数据结构)

NodeSet.clear();

VirusNode\* current;

queue<VirusNode\*>tempqueue;//用队列暂存结点,后端输入前端输出

tempqueue.push(root);

int order = 0;

while (!tempqueue.empty())

{

current = tempqueue.front();

tempqueue.pop();

current->order = order;

order++;

NodeSet.push\_back(current);

for (int i = 0; i < N; i++) {

if (current->subVirus[i] != NULL)

tempqueue.push(current->subVirus[i]);

}

}

}

void VirusTree::DestroyTree(VirusNode\* current) {

if (current != NULL) {

for (int i = 0; i < N; i++) {

DestroyTree(current->subVirus[i]);//遍历整棵树,将节点删除

}

delete current;

current = NULL;

}

}

5、算法说明

/\*关于LCA算法的解释\*/

/\*queue<node\*>ancestor;

void FindAncestor(node\* CurrentVirus, node\* PreVirus) {//在线算法，当数据很大时效率很低

if (CurrentVirus != NULL) {

if (CurrentVirus == PreVirus) {

ancestor.push(PreVirus);

}

FindAncestor(CurrentVirus->PreVirus, PreVirus);

}

}

node\* VirusTree::GetNearestAncestor(node\* virus1, node\* virus2) {

stack<node\*>temp;

temp.push(virus1);

while (temp.top() != NULL) {

node\* pre = new node;

pre = temp.top(); temp.pop();

FindAncestor(virus2, pre);

temp.push(pre->PreVirus);

}

return ancestor.front();

}\*/

/\*多个沙箱的实现\*/

/\*int min(int a, int b) {

if (a < b)return a;

else return b;

}

void SwitchSandBox(SandBox A, SandBox B) {//沙箱之间人员流动

for (int i = 0; i < min(A.GetHumanSet().size(), B.GetHumanSet().size()) / 10; i++) {

int randomA = rand() % A.GetHumanSet().size();

int randomB = rand() % B.GetHumanSet().size();

B.GetHumanSet().push\_back(A.GetHumanSet()[randomA]);

A.GetHumanSet().push\_back(B.GetHumanSet()[randomB]);

A.GetHumanSet().erase

(A.GetHumanSet().begin() + randomA, A.GetHumanSet().begin() + randomA + 1);

B.GetHumanSet().erase

(B.GetHumanSet().begin() + randomB, B.GetHumanSet().begin() + randomB + 1);

}

//由于时间原因,交换沙盒具体算法用文字替代

合并病毒树与合并病毒库:

1.本项目中,沙箱操作/人类结构的操作/并查集的操作全部是基于树的结构进行

如果人员流动,则需要引入树的添加与删除操作(基于判断病毒宿主是否全部交换的基础上)

且两棵不同的树的合并只是基于部分结点,因此生成的树有交叉或无交叉,需要判断;

2.由于算法设计时不同沙箱病毒树的随机性,

所以合并病毒树与病毒库的工作是基于大量查找的基础上,以保证病毒结点的不重合;

3.对于存活人数不同的沙箱,交换的人数也要有设定

4.交换后的病毒树由于有查找根结点的操作,可以在原有病毒树上实现

5.需对每个人类及病毒进行遍历,以判断所有结点情况

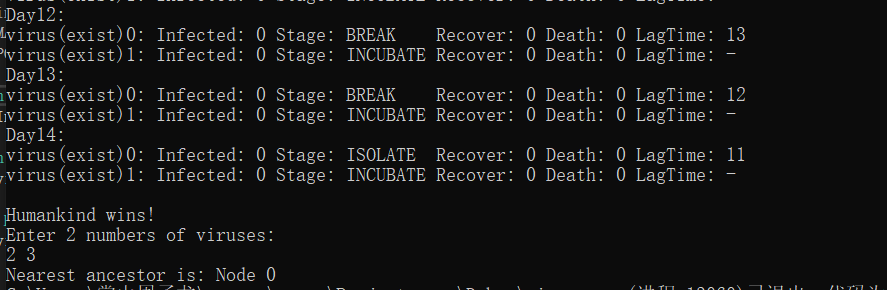
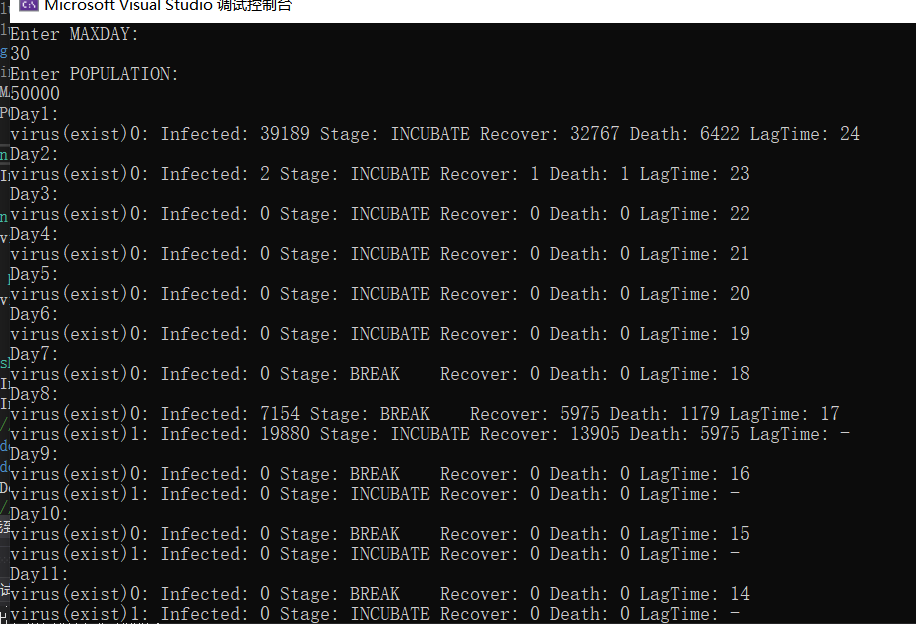
}\*/

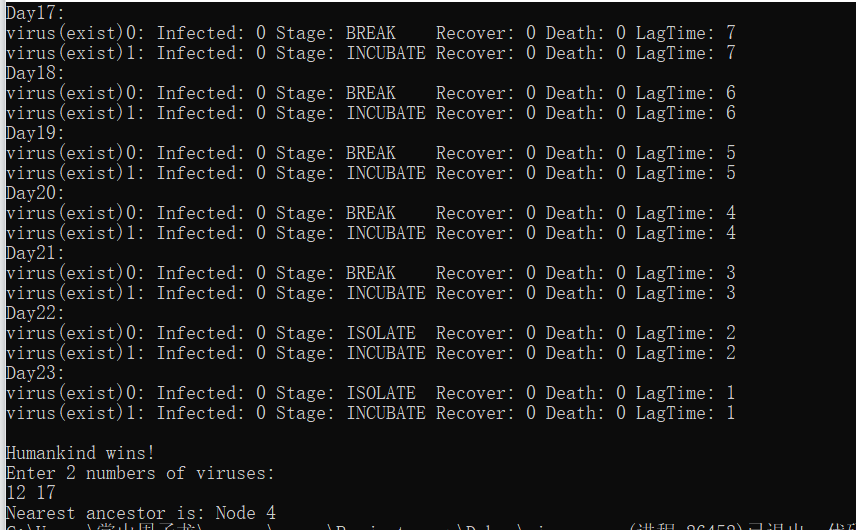
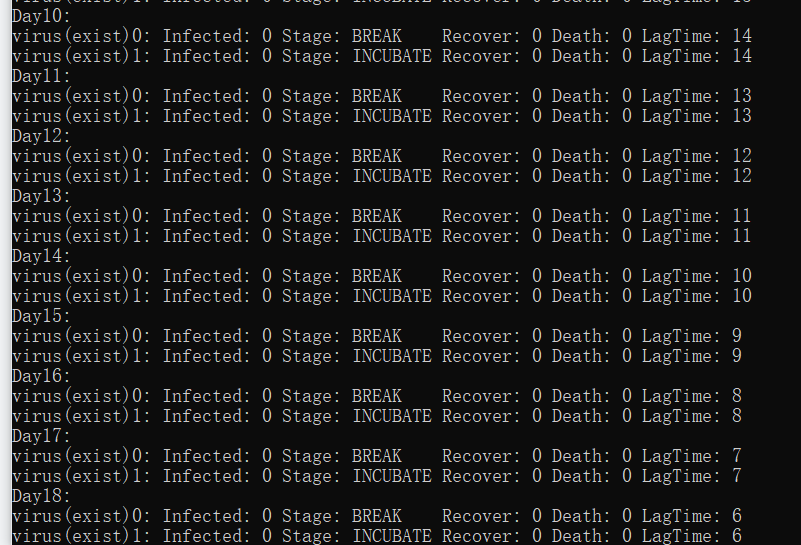
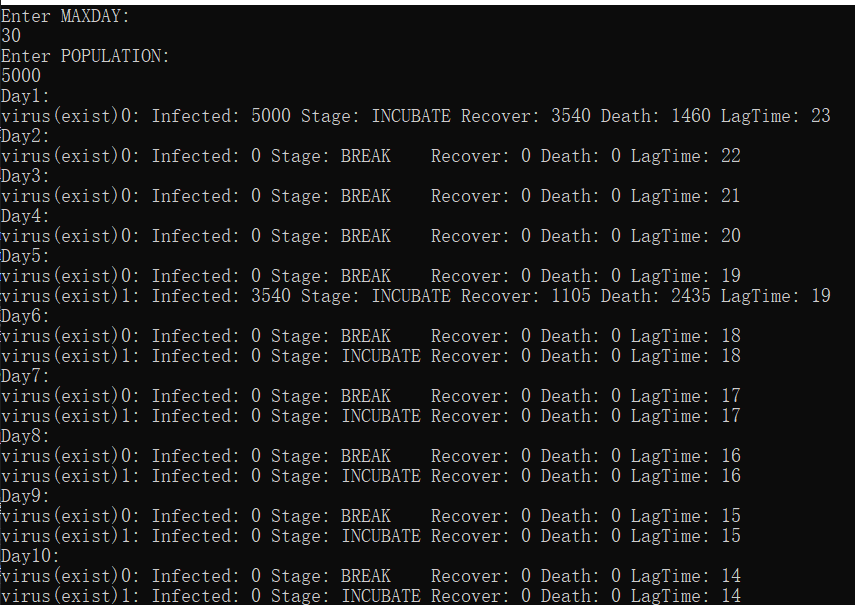
**四、 调试分析**

**1调试结果**









**2时间复杂度分析：**（设结点数为O（n）大小，人口数为O（m）大小）

**2.1复杂度为O（n）的程序：**

currentNodeSet（**O（n）**）：ifSucc（）、Clkdiv（）、Variation（）、

NodeSet（**O（n）**）：SaveEveryNode（）

**2.2 复杂度为O（m）的程序：**

HumanSet（**O（m）**）：Vaccine（）、Infect（）、SetHuman（）、SwicthState（）、Recover（）、Kill（）

**2.3 复杂度为其他的程序：**

main（）：**O（n\*log（n））**、SwitchSituation（）：**O（m\*n）**、GetNearestAncestor（）：**O（n+q）**

总的来说算法复杂度不超过O（m\*n）；

**3对模块调试存在的思考**

**3.1树的结构的选择**

最后选择的是普通树的结构，因为这样的结构稳定，遍历时方便查找，尽管有浪费空间这一缺点，但是在优化过程中将这一缺点转换为结点变异的稳定型，即先生成树再变异，便可以稳定的将未变异与已变异的结点相同处理，节省代码复杂度，且这样的结构有利于对结点的查找与输出，例如在存储结点时就可以利用普通树结构层次遍历的结果进行编号，输出最近公共祖先时便可以通过结点序号快速查找并输出。

**3.2存储结构的选择**

最后选择用的最多的结构是vector，因为vector是动态存储向量，一是方便插入与删除，并且可以动态调配空间，二是可以方便通过序号调用类里面的成员。这样的结构对于需要经常处理添加与删除操作的程序十分高效。

**3.3概率及传染率的实现**

利用随机数，且每次新生成一个输出时都要调用srand（）函数，对种子进行更新以保证每次输出的结果是完全随机的；另外，使用了Gauss函数，实现了正态分布随机数的生成，使得大部分的数据会落在期望的周围，最终使得统计结果可以收敛到最初设置的double类型的传染率。

**3.4最近公共祖先算法的实现**

一开始利用在线算法简单遍历，得到的结果复杂度很高，在网上查询资料以后了解到tarjan算法，可以在遍历时就利用并查集的性质将结点存储，只需调用遍历时的存储结果便可以实现最近公共祖先算法。

**3.5调试时，对于人类结构中mark数组的更新问题**

由于遍历时只用遍历已经变异的结点，所以在存储所有树上的结点之后不一定能够实时更新mark数组，于是，将更新结构移动至每次遍历树结点时，便可解决超出存储范围的问题。

**3.6循环调用结构的问题**

在实现沙箱时，定义的结构中Human包含了VirusNode结构，同样VirusNode也包含了Human结构，于是加上保护结构#ifndef，#define，#endif，还有提前声明，解决了该问题。

**4算法的改进**

**4.1查找算法问题：**由于在查找结点时以及对结构进行操作时都是简单的使用遍历操作，很浪费运行空间与时间，可以增加标记操作，对于具有某一性质的结构进行单独存储，可以节省查找操作的空间与时间，并且在操作完之后进行删除操作，便可以节约很大的运行空间。

**4.2存储结构的问题：**对于病毒结点的存储，简单的使用的是N叉树的定义，并且在加入结点以及删除结点时没有利用到树的优势，即复杂度为logn量级，而是简单利用数组进行遍历，并且对序号的数学关系没有很好的利用；可以将存储结构改进为平衡二叉树或堆等具有一定特殊性质的树结构，在调用结点时可以利用树和结点本身的性质来减少运行空间与时间。

**五、课程设计总结**

**1课程设计的收获**

本次课程设计让我对项目设计的理解更加深入，在结构方面让我了解到一个完整项目的设计流程是什么样的，在算法方面让我对更加快速、简洁的算法有了更高的追求，除此而外，通过本次课程设计，我还积极在网络及书籍中寻找资料帮助自己解决一些语法上的问题，对我的程序设计学习帮助很大。总的来说，本次课程设计让我对程序设计有了更高层次的理解。

**2遇到问题和解决问题的思考**

在设计模块时，遇到问题之后第一反应就是如何将设计变得更加合理，更加符合现实情况，在对于设计的不断调整过程中，我不断调整数据的结构以及范围试图让数据更加合理与可实现。在解决问题的过程中，不断调试代码，将溢出的空间以及逻辑错误不断修正，最终使得整个程序能够正常且理想的运行。

**3程序调试的体会**

首先，对于断点的设置技巧有了更深刻的理解，应该设置在关键调试输出处，其次，对于常见问题的解决方案也有了新的体会，例如对于溢出空间的处理就可以不断重置与调整以达到消除溢出错误的目的。本次课程设计对于我对程序调试的技巧有了极大的启发与帮助。

**4对数据结构课程的认识**

首先对于专业知识，数据结构这门课程对于一些结构的介绍对之后的学习帮助很大，不论是工作还是学习，本学期新学的一些存储结构都可以在算法设计中节省很多空间，提高很多效率；另外对于思维方面，数据结构这门课程拓展了我对于算法运行空间与时间的认识，让我了解到了算法设计的一些思维与逻辑，从而在以后的程序设计中能够更加系统的考虑到一些算法上所带来的问题与挑战。

**六、参考资料**

并查集的实现（参考了并查集操作的算法）：

https://blog.csdn.net/YSJ367635984/article/details/113504723?spm=1001.2101.3001.6650.1&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link&utm\_relevant\_index=2

Tarjan算法的实现：

<https://www.cnblogs.com/JVxie/p/4854719.html>

随机数的实现（Gauss函数）（参考了正太分布随机数的实现算法）：

https://blog.csdn.net/geter\_CS/article/details/86592639?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522164078329816780271985645%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\_id=164078329816780271985645&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-4-86592639.pc\_search\_insert\_es\_download\_v2&utm\_term=c%2B%2B%E9%9A%8F%E6%9C%BA%E6%95%B0&spm=1018.2226.3001.4187