无题

一 DFS和BFS

深搜通过栈来实现，从给定节点开始访问，直到该分支结束。而所有通过栈来实现的代码都可以通过递归来隐式的维护一个栈。一般情况下需要一个标记数组（visit）来配合使用，防止重复访问一个节点。

代码大致如下：

void DFS(int x)

{

visit[x] = 1;

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

{

if (visit[x][i] != 1&&map[x][i])

{

DFS(i);

}

}

}

解决的问题：遍历（太多了），寻找最深分支（深入虎穴），判断连通分支（红色警报）。

广搜，也叫层序遍历，常用于树的层序遍历。与深搜对应的，应用队列的来进行实现。

代码如下：

void BFS(int x)

{

queue<int> que;

que.push(x);

while (!que.empty())

{

int sz = que.size();

for (int i = 0; i < sz; i++)

{

int cur = que.front();

que.pop();

for (int j = 0; j < N; j++)

{

if(map[cur][i])

que.push(map[cur][j]);

}

}

}

}

解决问题：树的层序遍历（玩转二叉树，树的遍历），找出多个层次最深的节点（小字辈）

二 并查集

建立一个数组pre[N]，其中pre[i]的值就为i的父辈。通过find函数，来寻找传入函数的节点的最高祖先，可以在find函数加入路径优化。通过uinon函数将两节点合并到一个祖先下。

代码如下：

int pre[N];

void init()//初始化

{

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

pre[i] = i;

}

int find(int x)

{

if (pre[x] == x)

return x;

// while (pre[x] != x)

// {

// x = pre[x];

// }

// return x;循环访问，无路径优化

if (pre[x] != x)

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

//递归，有路径优化

}

void unio(int x, int y)

{

x = find(x);

y = find(y);

if (x == y)

return;

pre[x] = y;//x的上级改为y

}

解决问题：解决元素之间具有传递性，然后求解给定元素之间的关系的问题，实际题目中表现为朋友的朋友是朋友。（家庭房产，部落，排座位）。

三 结构体

在结构体的定义中重载比较运算符，再利用algorithm库的sort的函数按照给定标准进行排序。

代码如下：

const int N = 1e3 + 5;

struct S {

int id;

int first;

int second;

bool operator <(const S& p) const

{

//根据题目要求

if (first == p.first)

{

return second < p.second;

}

return first < p.first;

}

};

struct S s[N];

sort(s, s + N);

解决问题：利用结构体进行模拟，同时需要按照结构体内某一数据进行排序。（月饼，抢红包，名人堂）

四 堆

用优先队列实现，包含头文件queue和functional（greater和less函数）

push入堆，pop出队，top返回队头元素

priority\_queue<int, vector<int>, greater<int>> pq;

四 大数问题

大数加法

用cnt来计算进位，从后往前计算.

注意：1虽然加法进位只能为1但是计算进位时也应用除法不能直接减10；2形参可以使用引用的方式来减少空间消耗但是得确保后续计算中用不到s1和s2.

代码如下：

string jia(string s1,string s2)

{

if(s1.size()<s2.size())

{

string temp=s1;

s1=s2;

s2=temp;

}

int i,j,cnt=0;

for(i=s1.size()-1,j=s2.size()-1;j>=0;j--,i--)

{

int temp=s1[i]-'0'+s2[j]-'0'+cnt;

cnt=temp/10;

temp%=10;

s1[i]=char(temp+'0');

}

for(;i>=0&&cnt;i--)

{

int temp=s1[i]-'0'+cnt;

cnt=temp/10;

temp%=10;

s1[i]=char(temp+'0');

}

if(cnt)

{

s1=char(cnt+'0')+s1;

}

return s1;

}

大数乘法：

注意：

1 因为乘法有累加的过程，所以加法得先进行实现。

2 每次计算一个数字乘另外一串数字对每次的结果进行加0累加。

3 乘法同加法类似从后往前用cnt来控制进位但cnt的值不在固定为0和1了

string cheng(string s1,string s2)

{

string res="0";

for(int i=s1.size()-1;i>=0;i--)

{

int cnt=0;

string s="";

for(int j=s2.size()-1;j>=0;j--)

{

int temp=(s1[i]-'0')\*(s2[j]-'0')+cnt;

cnt=temp/10;

temp%=10;

s=char(temp+'0')+s;

}

// cout<<s<<endl;

if(cnt)

{

s=char(cnt+'0')+s;

}

for(int k=0;k<f;k++)

s+='0';

// cout<<s<<endl;

res=jia(res,s);

}

return res;

}

大数除法（不完全体）：

从前往后

注意：

1不够借位得补零，首位不补，判断条件为res非空；

2 cnt的变化

代码如下：

string div(string s,int num)

{

int cnt=0;

string res="";

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

{

int temp=s[i]-'0'+cnt\*10;

if(temp/num!=0)

{

cnt=temp%num;

temp/=num;

res+=char(temp+'0');

}

else

{

cnt=temp;

if(res!="")

res+='0';

}

}

if(cnt)

return "-1";

return res;

}

最短路径问题

1 dijk

注意：

邻接矩阵，单源点，动态规划

①初始化时将dis数组初始为inf，将dis[start]改为0。这样第一遍循环时就可将dis数组的值改为与start的距离，所以循环应该执行n而非n-1次；

②最后的通过x点来松弛dis数组时用多目运算符，好看一点.

代码如下：

void dijk()

{

init();

dis[start] = 0;

for (int i = 0; i < n; i++)

{

int x, min = inf;

for (int j = 0; j < n; j++)

{

if (!visit[j] && min > dis[j])

{

x = j;

min = dis[j];

}

}

visit[x] = 1;

for (int j = 0; j < n; i++)

dis[j] = dis[j] > dis[x] + map[x][j] ? dis[x] + map[x][j] : dis[j];

}

}

Dijk可以在使用邻接表和优先队列进行优化（减少了内存大小）

注意：

①数组g[i]的意思为：与第i个节点相关联的边，意味只有在单向路时使用才能最大程度的优化空间消耗。

②每次通过向优先队列中加入E(v,dis[v])一条新的经过松弛后的边，来实现dis数组的更新，节点v原有的数据并不会被删除依旧储存与队列中，只是不会再被调用了。

struct E{

int to;

int w;

bool operator<(const E& p) const

{

return w < p.w;

}

E(int x, int y)

{

to = x;

y = w;

}

}edge[N];

vector<E> g[N];//存放邻接表

void dijkster(int s)

{

for (int i = 0; i < n; i++)

dis[i] = inf;

dis[s] = 0;

priority\_queue<E> que;

que.push(E(s, 0));

while (!que.empty())

{

E cur = que.top();

que.pop();

int u = cur.to;

for (auto e : g[u])

{

int v = e.to;

if (dis[v] > dis[u] + cur.w)

{

dis[v] = dis[u] + cur.w;

que.push(E(v, dis[v]));

}

}

}

}

2 Floyd

邻接矩阵，多源点，贪心

注意：哈哈哈

代码如下：

void floyd()

{

init();

for (int k = 0; k < n; k++)

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

if (map[i][j] > map[i][k] + map[k][j])

map[i][j] = map[i][k] + map[k][j];

}

3 Bellman-Ford——解决负权边

邻接边，验证有无负环（正换）

大概原理：

和dijk大同小异，相同点在于都维护了一个dis数组来保存距离起点的距离，在进行松弛操作。不同点在于dijk通过点来进行松弛而Bford通过边来进行松弛，并且无视边的权值直接进行n-1次松弛。

判断有无负环的大概原理：

一个图的最短路径应该至多包含n个顶点，若进行了n-1次松弛操作还能进行松弛的话说明图中存在负环。

代码如下：

struct {

int start;

int end;

int w;

}edge[N];

bool bellmanford()

{

int dis[N];

for (int i = 0; i < n; i++)

dis[i] = inf;

dis[start] = 0;

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < edge\_size; j++)

{

if (dis[edge[j].end] > dis[edge[j].start] + edge[j].w)

dis[edge[j].end] = dis[edge[j].start] + edge[j].w;

}

}

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < edge\_size; j++)

{

if (dis[edge[j].end] > dis[edge[j].start] + edge[j].w)

return true;

}

}

return false;

}

4 spfa算法

邻接矩阵下的spfa，注意：

①visit数组表示该节点是否已在队列中，防止重复入队；

②cnt数组的初始化。

int cnt[N];//负环的判断

bool spfa(int s)

{

for (int i = 0; i < n; i++)

{

cnt[i] = 0;

dis[i] = inf;

visit[i] = false;

}

dis[s] = 0;

visit[s] = true;

cnt[s] = 1;

queue<int> que;

que.push(s);

while (!que.empty())

{

int cur = que.front();

que.pop();

visit[cur] = false;

for (int i = 0; i < n; i++)

{

if (dis[i] > dis[cur] + map[cur][i])

{

dis[i] = dis[cur] + map[cur][i];

if (!visit[i])

{

if (!cnt[i] > n)

return true;

que.push(i);

}

}

}

}

return false;

}

数组模拟邻接表的spfa，注意：

①使用之前应在主函数中调用init函数来初始化head数组；

②add操作是增加一条边，用head数组来模拟链表构造邻接表；

int head[N];

struct{

int to;

int w;

int next;//下一条相邻边

} edge[N];

int tot = 0;

void init()

{

memset(head, -1, sizeof(head));

}

void add(int x,int y, int w)

{

edge[tot].to = y;

edge[tot].w = w;

edge[tot].next = head[x];

head[x] = tot++;

}

bool spfa(int s)

{

int cnt[N];

memset(cnt, 0, sizeof(cnt));

memset(dis, inf, sizeof(dis));

memset(visit, 0, sizeof(visit));

dis[s] = 0;

visit[s] = 1;

cnt[s] = 1;

queue<int> que;

que.push(s);

while (!que.empty())

{

int cur = que.front();

que.pop();

visit[cur] = 0;

for (int i = head[cur]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (dis[v] > dis[cur] + edge[i].w)

{

dis[v] = dis[cur] + edge[i].w;

if (!visit[v])

{

if (++cnt[v] >= n)

return true;

visit[v] = 1;

que.push(v);

}

}

}

}

return 0;

}

用vector数组模拟的邻接表的spfa（和dijk的邻接表一样）：

注意：和之前用dijk中的一样，第一次将起点和权值0（edge(s,0)）作为一条边放入来达到初始化dis的目的

struct edge{

int to;

double w;

edge(int x,double y)

{

to=x;

w=y;

}

};

bool spfa(int s)

{

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

{

dis[i]=0;

cnt[i]=0;

vis[i]=0;

}

dis[s]=10;

cnt[s]=1;

vis[s]=0;

queue<edge> que;

que.push(edge(s,0));

while(!que.empty())

{

edge cur=que.front();

que.pop();

int u=cur.to;

vis[u]=0;

for(int i=0;i<g[u].size();i++)

{

edge e=g[u][i];

int v=g[u][i].to;

if(dis[v]<dis[u]\*e.w)

{

dis[v]=dis[u]\*e.w;

if(!vis[v])

{

if(++cnt[v]>=n)

return true;

vis[v]=1;

que.push(g[u][i]);

}

}

}

}

return false;

}

5最小生成树

Prim算法

注意：和dijk最短路径一样，区别就在于dis数组存放的是边的权值不需要累积，每次只需将最小的边取出就行。

int prim()

{

int res=0;

for(int i=0;i<n;i++)

{

dis[i]=inf;

vis[i]=0;

}

dis[0]=0;

for(int i=0;i<n;i++)

{

int u,min=inf;

for(int i=0;i<n;i++)

{

if(!vis[i]&&min>dis[i])

{

min=dis[i];

u=i;

}

}

vis[u]=1;

res+=min;

for(int i=0;i<n;i++)

{

if(!vis[i]&&dis[i]>map[u][i])

dis[i]=map[u][i];

}

}

return res;

}

Kruskal算法

注意：

①用并查集来检验是否会构成回路，每次加边时都将边的起点和终点unio，这样保证了已使用的且相连的边都有一个共同父辈。算法执行完后每个点都有共同父辈。

②用cnt来记录边数不含非负权边时最小生成树只能有n-1条边，以此作为结束条件

③优先队列默认为大顶堆。重载时应反着来。

struct edge{

int start,to,w;

bool operator<(const edge& p) const

{

return w>p.w;//优先队列默认大顶堆

}

edge(int x,int y,int v)

{

start=x;

to=y;

w=v;

}

};

int n;

int pre[N];

priority\_queue<edge> que;

int find(int x)

{

if(x==pre[x])

return x;

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

}

void uion(int x,int y)

{

x=find(x);

y=find(y);

if(x!=y)

pre[x]=y;

}

int kul()

{

int res=0,cnt=0;//边数

int p=que.size();

for(int i=0;i<p;i++)

{

edge cur=que.top();

que.pop();

int start=cur.start;

int end=cur.to;

if(find(start)!=find(end))

{

uion(start,end);

res+=cur.w;

cnt++;

}

if(cnt==n-1)

break;

}

return res;

}