АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА ЛАБИРИНТ

Графът, описан в условието на задачата, очевидно е дърво, затова броят на ребрата му е $M = N - 1$. Тривиално решение на задачата **в случай на неголямо** N е да намерим разстоянията от всеки връх до всеки от останалите върхове, да оформим таблица, в която за всеки връх i е записан по един връх на растояние 1, 2, …, d_i, където d_i е максималното разстояние от i до някой друг връх на дървото. Тогава отговорът на всяка заявка ще изисква константно време.

Ако за намирането на разстоянията използваме алгоритъма на Флойд, тогава сложността на решението ще бъде $O(N^3 + Q)$. Освен това, за табулирането на разстоянията ще е необходима таблица с размер $O(N^2)$, което очевидно е неприемливо за големи N.

Една възможност да подобрим бързодействието на такъв алгоритъм (но не и необходимата памет) е да използваме за табулиране на разстоянията обхождане в ширина – по едно обхождане за всеки връх. Тъй като всяко обхождане в ширина изисква $O(M) = O(N)$ стъпки, сложността на решението ще падне до $O(N^2 + Q)$, но заради необходимостта от много памет, такова решение също не може да разчита на много висока оценка.

Очевидно табулирането на разстоянията е основна пречка за получаване на добро решение. Да се откажем от табулиране. Елементарното решение в такъв случай е, за всяка заявка да извършим по едно обхождане в ширина, като го прекратим в момента, в който стигнем до връх на желаното разстояние. За такова решение е необходима много по-малко памет – $O(N)$, а броят на стъпките ще бъде $O(QN)$. Ако броят на заявките значително надвишава броя на върховете, можем да ускорим това решение до $O(N^2 + Q.\lg Q)$, като предварително сортираме заявките за време $O(Q.\lg Q)$, първо по съдържащия се в тях връх, а при еднакви върхове по разстоянията в нарастващ ред, и след това изпълним по едно търсене в ширина за всеки от върховете. Получените резултати трябва да се подредят по реда по който са били съответните им заявки, затова преди сортирането, във всяка заявка трябва да се добави и поредният ѝ номер в редицата от заявки. Такова решение не е интересно в случая, тъй като броят на заявките може да се счита равен на броя на върховете. Освен това, такова решение е възможно само ако можем да получим предварително всички заявки. Ако задачата се зададе в *интерактивна* форма – преди програмата да получи следваща заявка, трябва да даде отговор за предишната – такова решение е невъзможно.

Сега ще предложим един алгоритъм, който има подобно бързодействие, но може да работи и при *интерактивна* формулировка. Като начало, с две обхождания в дълбочина, с обща сложност $O(M) = O(N)$ намираме диаметъра на дървото. Разбиваме кореновото дърво D от второто обхождане в ширина (в него поне един диаметърът е път от някое от листата до корена), на непресичащи се по ребра пътища. Първият път е един диаметър, а след това за всеки, невлязъл в път връх v, построяваме пътя в D от v до най-близкия до него връх w, който е влязъл в някой от построените вече пътища. За всеки от пътищата, номерирани от 0 до K, запомняме *началото w, края v*, дължината му $d(w,v)$ и позицията $s(w,v)$ в масива, в който помним пътищата, където започва този път. За всеки влязъл в път връх v, без началото w, запомняме номера $p(v)$ на пътя, в който участва, началото w' на $p(v)$ и разстоянието $d(w',v)$.

Очевидно е, че когато обработваме заявката v,d, ако има връх на разстояние d от v, то има такъв връх в посока от v към запомнения диаметър Когато започнем да изпълняваме постъпващите заявки, построяваме най-дългата възможна редица v, w_1, w_2,…, w_r , където w_1 е началото на $p(v)$, w_i е началото на $p(w_{i-1})$, i =2, 3,…,r и $d(v, w_1)$ +

$d(w_1, w_2) + \ldots + d(w_{r-1}, w_r) \leq d$ – редица от „скокове“. Ако в последното условие имаме точно равенство – тогава търсеният връх е w_r. Ако не, търсеният връх е на разстояние $d - d(v, w_1) - d(w_1, w_2) - \ldots - d(w_{r-1}, w_r)$ от w_r по посока началото на неговия път. С едно изключение, ако w_r принадлежи на избрания диаметър, тогава последната стъпка може да се наложи да направим в обратната посока, т.е. към края на диаметъра, а не към началото му.
Разбира се, ако зададеното d е по-голямо от дължината на диаметъра, можем веднага да изведем 0, тъй като не може да има връх на такова разстояние от v. Ако d е на разстояние по-малко или равно на дължината на диаметъра, обаче, това не гарантира наличието на търсения връх. В този случай отсъствието ще установим чак при опита за последен скок, когато w_r се намира на диаметъра и в нито една от двете посоки няма връх на необходимото разстояние.

```
#include <stdio.h>
#define MAXN 20001

int nei[2*MAXN], start[MAXN][2];
int N, used[MAXN], q[MAXN], p[MAXN];
int paths[MAXN][4], conect[MAXN][3],brp,pstart,diam;


int BFS(int r)
{
   int i,x,y,max=0,maxv=r,b,e;
   for(i=1;i<=N;i++) used[i]= -1;
   q[0]=r;b=e=0; used[r]=0; p[r]=0;
   while(b<=e)
   {  x=q[b++];
      for(i=start[x][1];i<start[x+1][1];i++)
      {  y=nei[i];
         if(used[y]== -1)
         { used[y]=used[x]+1;p[y]=x;q[++e]=y;
           if(used[y]>max) {max=used[y];maxv=y;}
         }
      }
   }
   return maxv;
}

void build_paths(int to)
{
   int x,y,from,c=0;
   brp++;paths[brp][1]=to; x=to;
   while(used[x]!= -1) { x=p[x];c++; }
   from=x;x=to;paths[brp][2]=c;
   paths[brp][3]=pstart;
   y=pstart+c;pstart=y+1;
      nei[y--]=x;
   while(used[x]!= -1)
   {  conect[x][0]=brp;conect[x][1]=from;
      conect[x][2]=c--; used[x]= -1;
      x=p[x]; nei[y--]=x;
   }
   paths[brp][0]=from;
}

int findd(int v,int d)
{
    int p,con,lenf,lent;
```

```
    if(d>diam) return 0;
    while(conect[v][0]!=0&&d>conect[v][2])
    {  d-=conect[v][2];v=conect[v][1]; }
    if(conect[v][0]!=0||conect[v][0]==0&&d<=conect[v][2])
    { p=paths[conect[v][0]][3];return nei[p+conect[v][2]-d];}
    else
       if(conect[v][2]+d<=paths[0][2]) return nei[conect[v][2]+d];
       else return 0;
}

int main()
{
    int i,j,k,b,e,x,y,Q;

    scanf("%d",&N);k=0;
    for(i=1;i<=N;i++)
    {  scanf("%d",&start[i][0]);
       start[i][1]=k;
       for(j=1;j<=start[i][0];j++)
       { scanf("%d",&nei[k]);k++;  }
    }
    start[N+1][1]=k;
    b=BFS(1); e=BFS(b); diam=used[e];
   // postroyavane na patishta
    brp=0;
    paths[brp][0]=b;used[b]=-1;
    conect[b][0]=brp;conect[b][1]=b;conect[b][2]=0;
    paths[brp][1]=e; paths[brp][2]=used[e];
    paths[brp][3]=0;
    x=e;y=paths[brp][2];pstart=y+1;nei[y--]=x;
    while(used[x]!= -1)
    {  conect[x][0]=brp;conect[x][1]=b;
       conect[x][2]=used[x]; used[x]= -1;
       x=p[x];nei[y--]=x;
    }
    for(i=1;i<=N;i++)
       if(used[i] != -1) build_paths(i);
   // zayavki
   scanf("%d",&Q);
    for(i=1;i<=Q;i++)
    {  scanf("%d %d\n",&b,&x);
      e=findd(b,x); printf("%d\n",e);
    }
    return 0;
}
```

Да се опитаме да оценим сложността на този алгоритъм. Всяко от двете обхождания в ширина, както и построяването на пътищата, е със сложност $O(M) = O(N)$. За всяка от заявките, ще се наложи да се направят по няколко скока – в най лошия случай толкова, колкото е броят на пътищата на които сме разбили D. Броят на пътищата в никакъв случай не е по-голям от N, от където броят на стъпките за изпълнение на всички заявки е $O(QN)$ и това е сложността на решението в най-лошия случай. На практика, обаче, бързодействието на предложения алгоритъм е много по-добро – хипотезата ни е, че сложноста му е $O(Q.\lg N)$, но това предстои да се докаже.

Автор: Красимир Манев