ساختمان داده ها ۲۲۸۲۲

نیم‌سال اول ۱۴۰۱-۱۴۰۰

مدرس: مدرس: حسین بومري

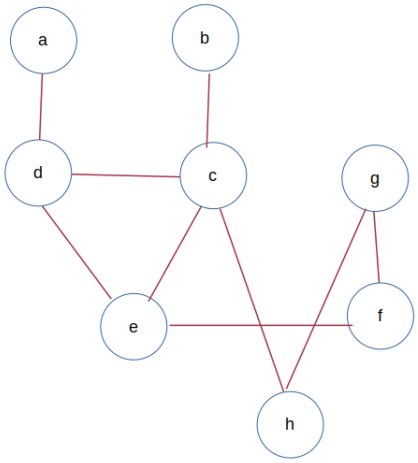
دانشکده علوم ریاضی

پیمایش‌های گراف

نگارنده: آرمیتا جلالیون

جلسه‌ی : بیست‌وپنجم

در جلسه‌ی گذشته بیش‌تر در مورد ساختمان داده و نحوه‌ی ذخیره‌سازی گراف صحبت کردیم. در این جلسه می‌خواهیم راجع به پیمایش‌های گراف صحبت کنیم.
می‌خواهیم در یک گراف غیرجهت دار مثل گراف زیر بفهمیم بین دو راس مسیر هست یا نه.



می‌توانیم برای این‌که بفهمیم بین دو راس a و f مسیر داریم یا نه لیستی درست کنیم که هر خانه‌ی آن متناظر با یک راس باشد.در ابتدا تمام خانه‌های آن‌را صفر قرار دهیم و هرگاه به راسی می‌رویم که قبلا آن را ندیده بودیم آن‌را یک می‌کنیم.
پس در نتیجه از راس a شروع می‌کنیم و به راس‌های مجاور در صورتی که قبلا دیده نشده باشند می‌رویم و خانه‌ی متناظر با آن راس را یک می‌کنیم و این‌کار را به صورت بازگشتی تا جایی ادامه می‌دهیم که یا خانه‌ی دیده نشده‌ای نماند و یا به راس مورد نظر برسیم.
در نهایت در این پیمایش به یک درخت می‌رسیم و به این پیمایش dfs یا search first depth نام دارد.
کد جاوای این الگوریتم:

```java
public class DFS {
    static int graph[][];

    public static void readGraph(Scanner scanner) {
        int n=scanner.nextInt();
        int m=scanner.nextInt();

        graph = new int[n][n];

        for(int i=0;i<n;i++) {
            for(int j=0;j<n;j++) {
                graph[i][j]=0;
            }
        }

        for(int i=0;i<m;i++) {
            int s=scanner.nextInt();
            int e=scanner.nextInt();

            graph[s-1][e-1]=1;
        }
    }

    public static void main(String[] args) throws FileNotFoundException {
        Scanner scanner=new Scanner(new File( s: "data.txt"));

        readGraph(scanner);

        boolean[] visited = new boolean[graph.length];
        for(int i=0;i<graph.length;i++) {
            visited[i]=false;
        }

        dfsMatrix(graph,  current: 0, visited);
    }

    public static void dfsMatrix(int[][] graph, int current, boolean[] visited) {
        visited[current] = true;
        for(int i=0;i<graph.length;i++) {
            if(!visited[i] && graph[current][i]!=0) {
                System.out.println(current+"->"+i);
                dfsMatrix(graph,i,visited);
            }
        }
        System.out.println("return from "+current);
    }
```

برای تحلیل زمانی این تابع باید در نظر داشته باشیم که تابع readGraph از اوردر $n^2 + m$ کار انجام می‌دهد.
برای تحلیل زمانی dfs ، به این نکته دقت می‌کنیم که به ازای یک راس حداکثر یک‌بار وارد آن می‌شویم و آن را به‌عنوان current در نظر می‌گیریم. چرا که هر بار visited آن‌را چک می‌کنیم؛ هم‌چنین به ازای هر راس همه‌ی راس‌ها چک می‌شود که اگر visited آن‌ها ۱ نباشد و با راس فعلی مجاور باشند واردشان شویم. در نتیجه این dfs از $O(n)$ قابل انجام است.
حالا فرض کنید همین کد را به نحوی تغییر دهیم که به‌جای ماتریس برای ذخیره‌ی گراف از n تا list array استفاده می‌کنیم که در هر list array مربوط به راس مورد نظر رئوس مجاور آن را اد می‌کنیم.
در این حالت readGraph ما از اوردر $n + m$ قابل انجام است.
برای تحلیل زمانی dfs در این حالت $o(m + n)$ می‌شود چرا که ما هر راس را نهایتا یک‌بار می‌بینیم و در for آن به‌اندازه‌ی تعداد همسایه‌هاfor می‌زنیم که در مجموع برای ما به اندازه‌ی m+n هزینه خواهد داشت.
این پیمایش که راجع به آن صحبت کردیم در هر مرحله تا جای ممکن عمیق می‌شد برای همین به آن search first depth گفتیم.
می‌توانیم نوع دیگری از پیمایش تعریف کنیم که از راس مورد نظر شروع می‌کنیم و سطح آن را صفر قرار می‌دهیم. در هر مرحله به تمام راس‌هایی که می‌توان از راس فعلی به‌آن رفت و از قبل ندیده باشیم می‌رویم و سطح آن‌ها را برابر با سطح راس فعلی+۱ قرار می‌دهیم.
مثلا در گرافی که در ابتدا رسم کردیم، سطح‌ها به صورت:

d=1
c , e =2
h,b,f=3
g=4

خواهند بود.
به این پیمایش bfs یا search Breadth-first می‌گوییم.
در آینده خواهیم دید که می‌توان یک کد مشابه را با دو ساختمان داده‌ی صف و پشته زد که یکی از آن‌ها کد bfs باشد و دیگری کد dfs.