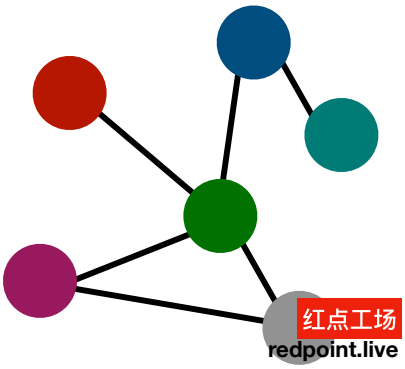
Javascript数据结构

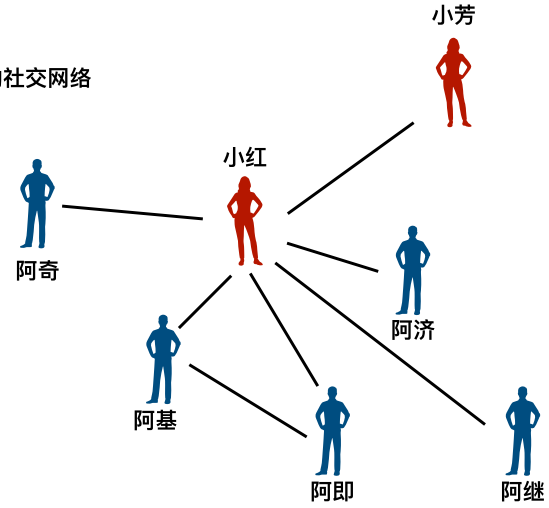
图

Skipper

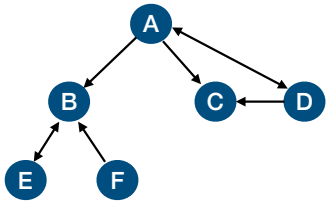


图是一种计算机中使用广泛的结构

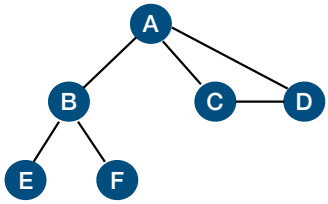
比如：表示我们常用的社交网络



图一些基本概念

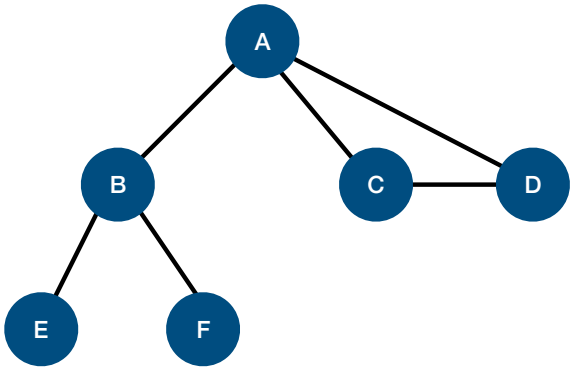


有向图



无向图

图的表示方式

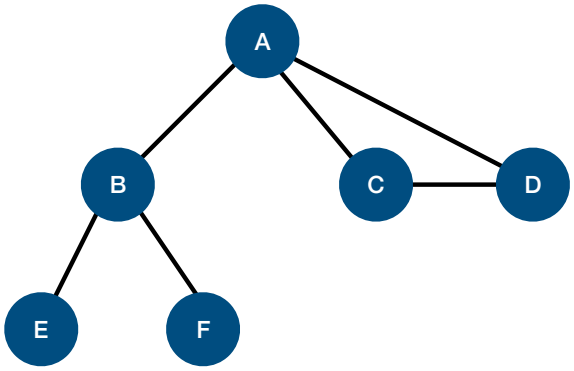


邻接矩阵

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	1	0	0
D	1	0	1	0	0	0
E	0	1	0	0	0	0
F	0	1	0	0	0	0

缺点：

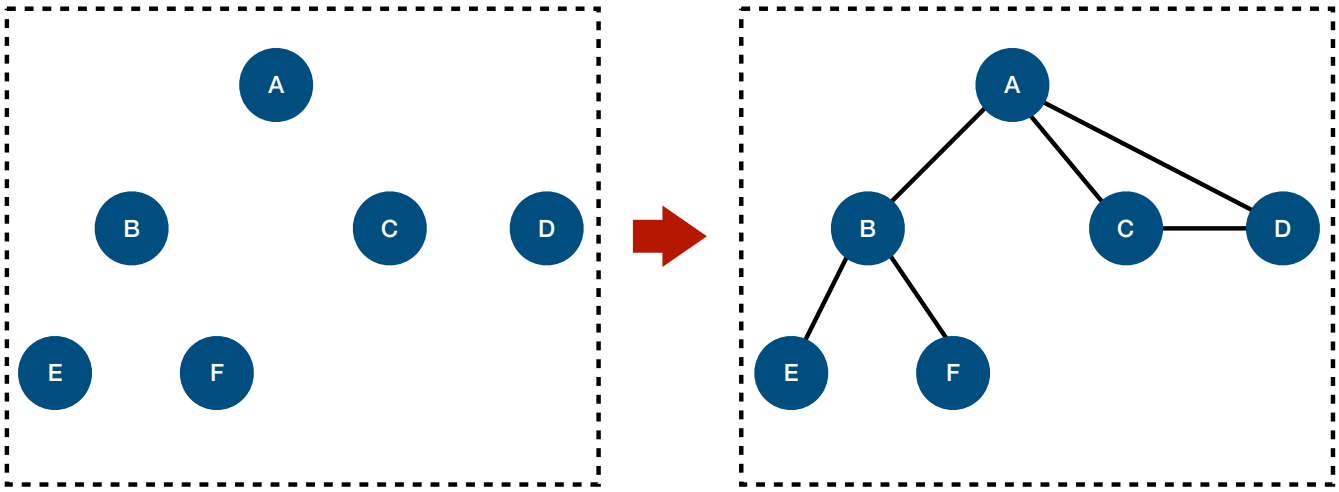
1. 非常浪费计算机内存
2. 添加和删除点很麻烦



邻接表

A	B C D
B	A E F
C	A D
D	A C
E	B
F	B

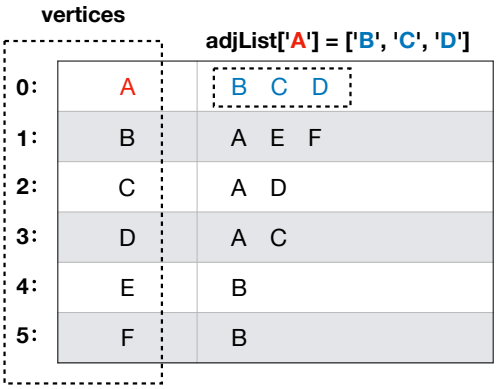
邻接表图实现方式



1. 添加顶点

2. 添加顶点之间的边

```
var Graph = function(){

    // 顶点
    var vertices = []
    // 边
    var adjList = {}

}
```



•vertices以数组形式存储每个顶点

vertices = [A B C D E F]

•adjList以对象形式存储每个顶点包含的边

```
adjList = {
    A : [ ],
    B : [ ],
    //…
}
```

```
var Graph = function(){

    // 顶点
    var vertices = []
    // 边
    var adjList = {}

    this.addVertex = function(v){
        vertices.push(v)
        adjList[v] = []
    }

    this.addEdge = function(a, b){
        adjList[a].push(b)
        adjList[b].push(a)
    }
}
```

```
var g = new Graph()

g.addVertex('A')
g.addVertex('B')
g.addVertex('C')
```

vertices.push('A')
adjList['A']=[]

...

vertices

A	[ ]
B	[ ]
C	[ ]

```
g.addEdge('A','C')
```

adjList['A'].push('C')

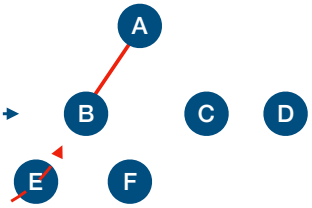
adjList['C'].push('A')

vertices

A	[C]
B	[ ]
C	[A]

```
var g = new Graph()
var vl = ['A','B','C','D','E','F']
vl.forEach(function(item){
    g.addVertex(item)
})

g.addEdge('A','B')
g.addEdge('A','C')
g.addEdge('A','D')
g.addEdge('B','E')
g.addEdge('B','F')
g.addEdge('C','D')
```

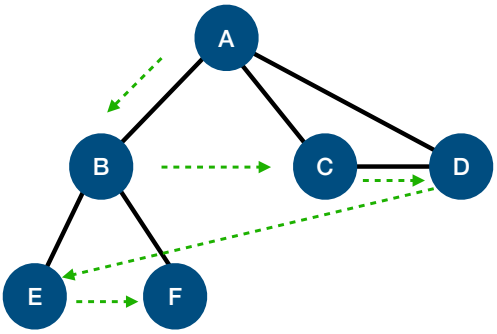
添加顶点：A B C D E F



添加 A和B 边缘

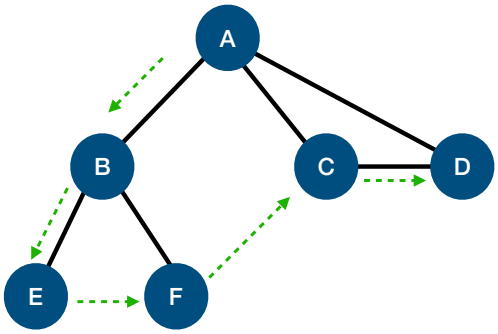
图遍历：广度优先和深度优先

广度优先



优先遍历图的横向

深度优先

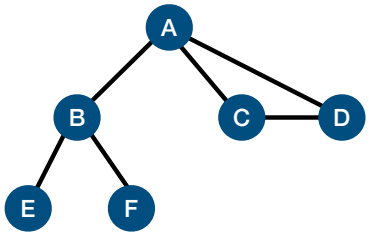


优先遍历图的纵向

图遍历基本思路

每一个节点有三种状态

- 未发现（尚未发现此节点）
- 已经发现（发现其他节点连接到此，但未查找此节点全部连接的节点）
- 已经探索（已经发现此节点连接的全部节点）



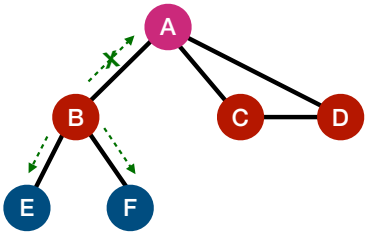
假设A为已探索

B、C、D为已发现

E、F为未发现

广度优先遍历流程如下

- 发现未发现节点后放在队列中，等待查找，并且标志为已发现
- 在队列中拿出已发现节点开始探索全部节点，并且跳过已发现节点
- 遍历完此节点后，将此节点标志为已探索
- 开始在队列中探索下一节点



例子

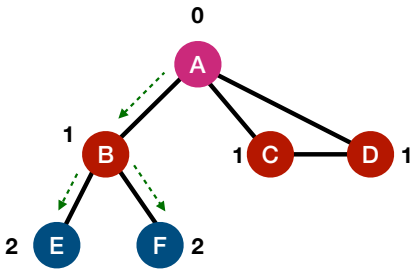
- 假设此时A已探索，则队列中存在已发现节点B、C、D
- A标记为已探索
- 开始探索队列中下一节点B
- B探索的全部节点包括A、E、F，但A已探索（或者已发现），跳过A
- 发现E、F节点，放入队列中
- B标记为已探索（可以操作B了）
- 开始探索队列中下一节点C
- ...

最终遍历顺序：A、B、C、D、E、F

广度优先遍历和最短路径问题

假设从A开始

1. 记录d(distance) = {A : 0 , B :1 , E : 2 …}
2. 记录回溯路径 pred = [A : null , B : 'A', E : 'B' , F : 'B'…]



- 每次探索新点后（如E），设置回溯点（B）
  pred['E'] = 'B'
- 每次探索新点后（如E），将该点距离（d['E']）设为回溯点（B）的距离加1
  d['E'] = d['B'] + 1

代码示范

广度优先

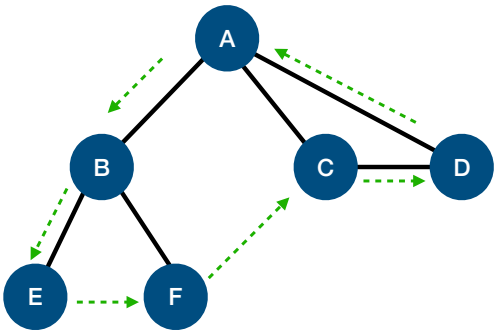
```
this.bfs = function (v, callback) {
    var color = initalizeColor()
    var queue = new Queue()
    queue.enqueue(v)

    while (!queue.isEmpty()) {
        var u = queue.dequeue()
        var n = adjList[u]
        color[u] = 'grey'

        for (var i = 0; i < n.length; i++) {
            var w = n[i]
            if (color[w] === 'white') {
                color[w] = 'grey'
                queue.enqueue(w)
            }
        }
        color[u] = 'black'
        if (callback) {
            callback(u)
        }
    }
}
```

带最短路径的广度优先

```
this.BFS = function (v) {
    var color = initalizeColor()
    var queue = new Queue()
    var d = []
    var pred = []
    queue.enqueue(v)
    for (var i = 0; i < vertices.length; i++) {
        d[vertices[i]] = 0
        pred[vertices[i]] = null
    }
    while (!queue.isEmpty()) {
        var u = queue.dequeue()
        var n = adjList[u]
        color[u] = 'grey'
        for (var i = 0; i < n.length; i++) {
            var w = n[i]
            if (color[w] === 'white') {
                color[w] = 'grey'
                d[w] = d[u] + 1
                pred[w] = u
                queue.enqueue(w)
            }
        }
        color[u] = 'black'
    }
    return {
        distance: d,
        predecessors: pred
    }
}
```

深度优先遍历

- 从某一节点开始查找，并且将自己标志为已发现
- 从此节点继续探索其全部节点，并且跳过已发现节点
- 遍历完此节点后，将此节点标志为已探索
- 递归返回，继续探索下一路径的最深节点



例子

- 假设此时A已发现，并且发现B、C、D节点
- 开始探索下一节点B
- B发现的全部节点包括A、E、F，但A已发现，跳过A
- 开始探索下一节点E
- E探索完毕，递归返回
- 继续探索下一节点F
- F探索完后，B标志为已探索，开始探索A的下一节点C
- ...

最终遍历顺序：E、F、B、D、C、A

代码示范：

```
var dfsVisit = function (u, color, callback) {
    color[u] = 'grey'
    var n = adjList[u]
    for (var i = 0; i < n.length; i++) {
        var w = n[i]
        if (color[w] === 'white') {
            dfsVisit(w, color, callback)
        }
    }
    color[u] = 'black'
    if (callback) {
        callback(u)
    }
}
this.dfs = function (v, callback) {
    var color = initalizeColor()
    dfsVisit(v, color, callback)
}
```