

FDS 算法精髓总结

--by JerryG

//个人对于 FDS 的一些粗浅理解

(标红为极有可能期末考的编程题)

1. 二叉树的遍历算法

(1) 前中后序遍历

比较简单的递归实现

🔑 **Tree Traversals** — visit each node exactly once

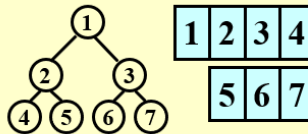
❖ Preorder Traversal

```
void preorder ( tree_ptr tree )
{ if ( tree ) {
    visit ( tree );
    for (each child C of tree )
      preorder ( C );
  }
}
```

❖ Postorder Traversal

```
void postorder ( tree_ptr tree )
{ if ( tree ) {
    for (each child C of tree )
      postorder ( C );
    visit ( tree );
  }
}
```

```
void inorder ( tree_ptr tree )
{ if ( tree ) {
    inorder ( tree->Left );
    visit ( tree->Element );
    inorder ( tree->Right );
  }
}
```



算法区别主要在于操作函数 `visit` 调用语句和递归调用语句的位置关系，本质上都是 DFS（后面图论部分有详细说明）。另外也可以通过迭代方法实现（写起来稍麻烦一点）。

(2) 层序遍历

❖ Levelorder Traversal

```
void levelorder ( tree_ptr tree )
{ enqueue ( tree );
  while (queue is not empty) {
    visit ( T = dequeue ( ) );
    for (each child C of T )
      enqueue ( C );
  }
}
```

层序遍历与上面几种遍历的根本区别在于层序遍历是经典的 BFS，并且是 based on queue 的 BFS。具体原理在图论一节有详细的说明，这里略过。

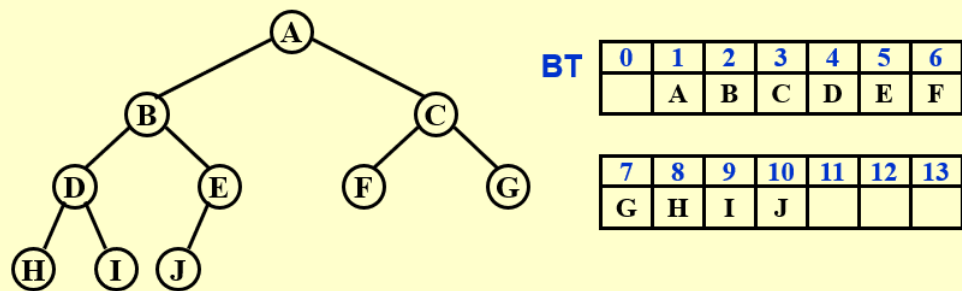
(3) 根据两个遍历序列还原树

这也是常见题目，关键在于确定根节点的位置，再递归解决左右子树。情况比较多，Leetcode 上对每一种情况都有对应的题目和详细的解答。

2. 优先队列/最小堆算法：

最小堆是一种非常重要的数据结构，它的特点是无序输入进去的元素，在输出时会按从小到大的顺序输出，里面用数组实现了一个抽象的二叉完全树。之所以叫堆，很可能就是因为二叉完全树长得很像一堆东西（稻谷）堆在一起，其主要特点是树的根节点的值永远是最小的那个。堆的优点在于查找最小值复杂度是 $O(1)$ ，缺点是元素进入（insert，或 Percolate up）和删除节点（根节点 DeleteMin，或者 Percolate down）的复杂度都是 $\log(N)$ 。综合来看，如果想对一个数组排序，那么将其元素依次入堆，最后依次 DeleteMin 即可得到一个有序序列，复杂度是 $N\log N$ ，较优，而且相比于其它 $N\log N$ 的排序方式，堆排序的优点是动态性比较好，DeleteMin 不会破坏堆的性质，可以边增减元素边排序，不用推倒重来。因此很多算法中涉及比较、排序、选最值的地方，通常可以使用堆优化。

❖ Array Representation : **BT[n + 1]** (BT[0] is not used)



【Lemma】 If a complete binary tree with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:

$$(1) \text{ index of } \text{parent}(i) = \begin{cases} \lfloor i/2 \rfloor & \text{if } i \neq 1 \\ \text{None} & \text{if } i = 1 \end{cases}$$

$$(2) \text{ index of } \text{left_child}(i) = \begin{cases} 2i & \text{if } 2i \leq n \\ \text{None} & \text{if } 2i > n \end{cases}$$

$$(3) \text{ index of } \text{right_child}(i) = \begin{cases} 2i + 1 & \text{if } 2i + 1 \leq n \\ \text{None} & \text{if } 2i + 1 > n \end{cases}$$

堆使用完全二叉树，最大的优点就是可以用数组存储表示，父节点和子节点的下标有固定的关系，极大减小复杂度。千万注意 PPT 中给出的是下标从 1 开始的情况，实际程序中如果从 0 开始，那么上面公式中所有等号右边 i 都要加 1。

堆数据结构中最重要的算法就是 Percolate，即过滤，分为向下过滤和向上过滤。添加元素对应于向上 Percolate up，即最开始把要添加的元素放在数组末尾，也就是二叉完全树最底下一行的最右侧，然后与父节点比较，如果比父节点小则与父节点交换位置，否则终止。

```

/* H->Element[ 0 ] is a sentinel */
void Insert( ElementType X, PriorityQueue H )
{
    int i;

    if ( IsFull( H ) ) {
        Error( "Priority queue is full" );
        return;
    }

    for ( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];

    H->Elements[ i ] = X;
}

```

Percolate up

Faster than swap

$T(N) = O(\log N)$

(注意 i 的+1 问题)

另一个就是 Percolate Down，即向下过滤，一般用于 DeleteMin 的情况，即向下过滤根节点。向下过滤比向上要复杂不少，因为它额外涉及了两点：

- (1) 判断下面是否还有子节点，是只有左节点还是左右都有，这里切忌不加判断直接与子节点比较导致内存越界！（堆最容易出错的地方）
- (2) 如果只有左节点，那当然跟它作比较，如果左节点更小交换即可。但如果左右两个子节点，且都小与它，此时必须再比较左右两个子节点哪个更小，跟更小的交换。

```

ElementType DeleteMin( PriorityQueue H )
{
    int i, Child;
    ElementType MinElement, LastElement;
    if ( IsEmpty( H ) ) {
        Error( "Priority queue is empty" );
        return H->Elements[ 0 ];
    }
    MinElement = H->Elements[ 1 ]; /* save the min element */
    LastElement = H->Elements[ H->Size-- ]; /* take it out and reset size */
    for ( i = 1; i * 2 <= H->Size; i = Child ) { /* Find smaller child */
        Child = i * 2;
        if ( Child != H->Size && H->Elements[ Child+1 ] < H->Elements[ Child ] )
            Child++;
        if ( LastElement > H->Elements[ Child ] ) /* Percolate one level */
            H->Elements[ i ] = H->Elements[ Child ];
        else break; /* find the proper position */
    }
    H->Elements[ i ] = LastElement;
    return MinElement;
}

```

Percolate down

3. 并查集 find 算法

并查集是一种高效的解决分类问题的算法。集合上的等价关系可以确定集合的一种分类，依次输入等价元素对，就可以通过并查集输出各个等价类。

并查集的数据结构是用一维数组 S 实现的森林，森林的每棵树都代表了一个等价类，刚开始时每个点是独立的，后面随着等价元素对的输入进行 union 操作逐渐生成森林。每一棵树的根节点，是这个等价类的代表元，用它来代表整个类，可以更换。注意这里的树是反向的，即边都是从子节点指向父节点。

Union 和 Find 函数是并查集的核心函数，而 Union 又是基于 Find 的。

Find(i) 返回节点 i 所在等价类的代表元，实现方法较为简单，向上追溯即可： $S[i]$ 是 i 的父节，一直往上，直到遇到 -1 。初始时数组 S 中全为 -1 ， -1 就代表自己就是等价类的代表元。

Union(i, j) 的功能是将节点 i, j 所在的等价类合并，即合并两棵树，具体的方法是先 find 到一个节点的根节点，然后让这个根节点的 S 的值等于另一个节点的根节点下标。

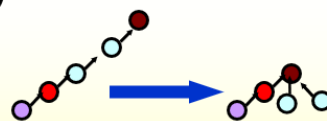
并查集有诸多优化方法。可以看到并查集主要时间消耗在于 Find 中追溯根节点的步骤，所以优化的核心自然是减少每颗树的 height，避免树太高，有三种方法。

方法一：Union-by-Size，好理解，就是合并时让矮的树合并到高的树上面，从而只有在两棵树等高时才会高度增加 1。这需要在生成树的同时记录树的高度，有一个非常完美的方法就是用根节点的 S 值来表示，并且为了跟有父节点的情况做区分，用负数表示，每次 Union 的时候凭借 $S[\text{Find}(i)]$ ， $S[\text{Find}(j)]$ 的绝对值大小判断高度，如果两棵树等高了，那么被合并到的那棵树的根节点 S 值减一。这样可以将树的平均高度维护在 $\log N$ 的水平。

方法二：Path-Compression，是一种比较奇怪的做法，在 Find 的时候改变树的结构，使之扁平化，效率比较高，图示和伪代码见 PPT。这里的 Find 代码 for 循环的流程最好记住，考试常考，容易写错。

§ 5 Path Compression

```
SetType Find ( ElementType X, DisjSet S )
{
    if ( S[X] <= 0 ) return X;
    else return S[X] = Find( S[X], S );
}
```



```
SetType Find ( ElementType X, DisjSet S )
{ ElementType root, trail, lead;
  for ( root = X; S[root] > 0; root = S[root] )
      ; /* find the root */
  for ( trail = X; trail != root; trail = lead ) {
      lead = S[trail];
      S[trail] = root;
  } /* collapsing */
  return root;
}
```

Note: Not compatible with union-by-height since it changes the heights. Just take “height” as an estimated rank.

方法三：Union-by-Rank

说的简单点就是 Union-by-Size 和 Path-Compression 一起用，这里其实有一点逻辑问题，就是 Path-Compression 的过程已经改变了树的高度，而由于反向树的特征，并不容易在这个过程中得到具体新树的高度并存入 $S[root]$ ，所以 Union-by-Size 时根节点的 S 值并不是记录的真正树的高度，而是“Rank”，一个比较虚的概念，也不重要，这里只需要知道两个一起用可行并且可以减少复杂度就行了。

图论算法

//图算法思想：BFS（广度优先搜索）和 DFS（深度优先搜索）


FDS 讲了很多图的算法，其中核心主线是两种算法思想，即 BFS 和 DFS。

BFS 形式上常常是“扩散”形态，逐步推移 queue 或者确定 known set，即已经求得某些关键量的点，最终扩散到终点。BFS 适用的问题通常由 greedy 的性质，并且图里一般都没有圈（有圈的话，题目性质最好能保证可以回避圈，如最短路径问题），实现形式可以通过 queue，也可以听过 known 数组来确定 known set，实现扩散。

DFS 思想是沿着一条路径深入探寻，不可行则退回，尝试其他最深路。DFS 在解决路径查找、生成树问题中很有用，在找到一条路径即可的情况下要比 BFS 效率要高。DFS 实现形式通常是递归，调用自身对应于深入探寻，return 则意味着退回来走其他路。

4. 图 BFS-拓补排序算法

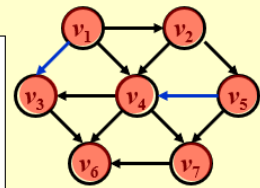
```
void Topsort( Graph G )
{
    int Counter;
    Vertex V, W;
    for ( Counter = 0; Counter < NumVertex; Counter ++ ) {
        V = FindNewVertexOfDegreeZero(); /*  $O(|V|)$  */
        if ( V == NotAVertex ) {
            Error ( "Graph has a cycle" ); break; }
        TopNum[ V ] = Counter; /* or output V */
        for ( each W adjacent to V )
            Indegree[ W ] -- ;
    }
}
```

 $T = O(|V|^2)$

基本思路是通过控制点的 indegree 来分析其层级高低。每次都找一个 indegree 为 0 的点（找不到说明由圈），列入排序列（开始为空）最后，然后把它连向的所有点入度都减 1，重复此过程。

Improvement: Keep all the unassigned vertices of degree 0 in a special box (queue or stack).

```
void Topsort( Graph G )            $T = O(|V| + |E|)$ 
{
    Queue Q;
    int Counter = 0;
    Vertex V, W;
    Q = CreateQueue( NumVertex ); MakeEmpty( Q );
    for ( each vertex V )
        if ( Indegree[ V ] == 0 ) Enqueue( V, Q );
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        TopNum[ V ] = ++ Counter; /* assign next */
        for ( each W adjacent to V )
            if ( -- Indegree[ W ] == 0 ) Enqueue( W, Q );
    } /* end-while */
    if ( Counter != NumVertex )
        Error( "Graph has a cycle" );
    DisposeQueue( Q ); /* free memory */
}
```



Indegree	
v ₁	0
v ₂	0
v ₃	0
v ₄	0
v ₅	0
v ₆	0
v ₇	0

队列优化的线性算法。每次将所有入度为 0 的点入队，然后依次出队并加入排序列，出队后把它连向的所有点入度减一，减一的同时判断其是否入度变为 0，如果是则入队。直至队列为空。

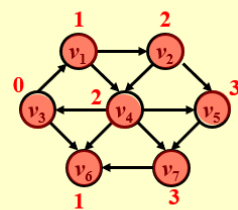
减少了复杂度的根本原因在于 FindNextZero，也就是找入度为 0 点的过程在给出队点的相邻点入度减一的过程同时完成了，而不必减完之后从所有点中一起查找判断 ($O(V)$)。

拓补排序是一种 BFS，只不过扩散入队的时候有减一后入度为 0 的条件。这个算法与树的层序遍历也比较相像。

5. 图 BFS-不带权最短路径算法(也可作为判断两点是否连通的算法)

Improvement

```
void Unweighted( Table T )
{
    /* T is initialized with the source vertex S given */
    Queue Q;
    Vertex V, W;
    Q = CreateQueue( NumVertex ); MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        T[ V ].Known = true; /* not really necessary */
        for ( each W adjacent to V )
            if ( T[ W ].Dist == Infinity ) {
                T[ W ].Dist = T[ V ].Dist + 1;
                T[ W ].Path = V;
                Enqueue( W, Q );
            } /* end-if Dist == Infinity */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
```



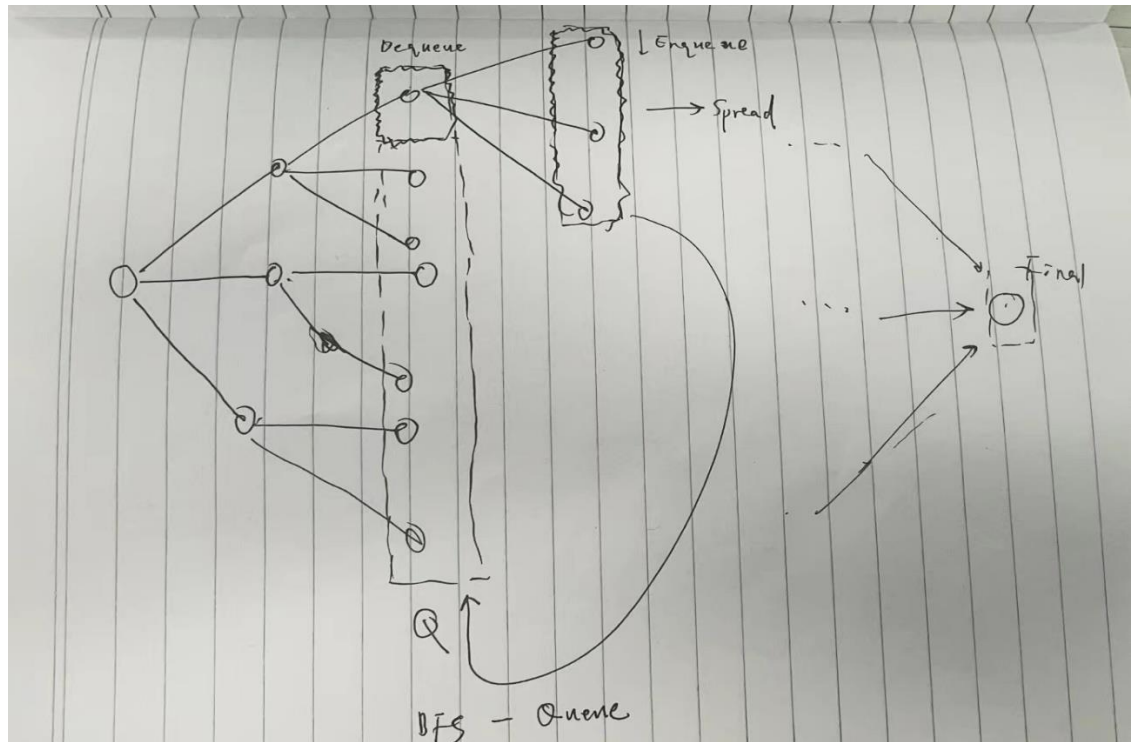
Dist Path	
v ₁	1 v ₃
v ₂	2 v ₁
v ₃	0 0
v ₄	2 v ₁
v ₅	3 v ₂
v ₆	1 v ₃
v ₇	3 v ₄

$$T = O(|V| + |E|)$$

与拓补排序的 BFS 思想类似，也是通过一个 queue 来实现 bfs。但是要设置一个 Dist 数组记录最短路径，初始设为正无穷，对于出队的点 V，把它相邻的、Dist 未设定的点的 Dist 设为 $\text{Dist}[V]+1$ ，然后入队。

若想输出路径也比较简单，只需要记录使得每个点的 Dist 取最小的路径中它的上一个节点即可，从而最终只需要从后向前不断追溯即可。

与拓补排序还有一点不同就是初始入队的不是所有入度为 0 的点，而是唯一的那个起点，比较好理解。可以看到用 queue 实现 BFS 还是有一些共性的，那就是扩散的时候通过队列“推进”的方式扩散。



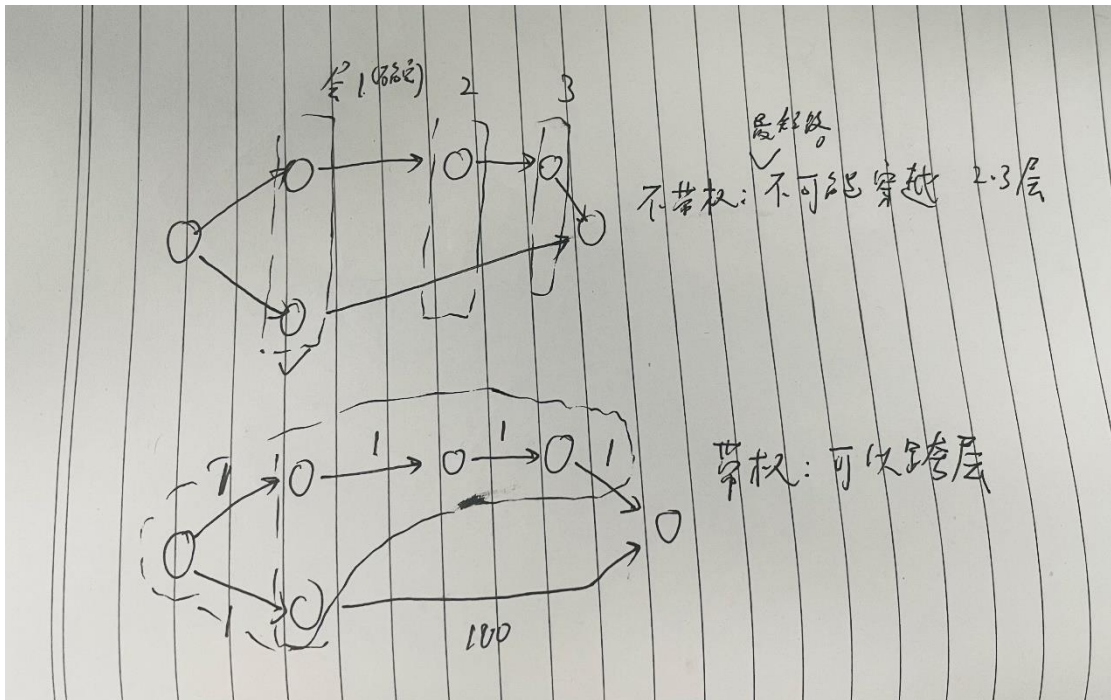
6. 图 BFS-Dijkstra 算法

```
void Dijkstra( Table T )
{ /* T is initialized by Figure 9.30 on p.303 */
  Vertex V, W;
  for ( ;; ) { /* O( |V| ) */
    V = smallest unknown distance vertex;
    if ( V == NotAVertex )
      break;
    T[ V ].Known = true;
    for ( each W adjacent to V )
      if ( !T[ W ].Known )
        if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
          Decrease( T[ W ].Dist to
              T[ V ].Dist + Cvw );
          T[ W ].Path = V;
        } /* end-if update W */
    } /* end-for( ;; ) */
  } /* not work for edge with negative cost */
```

这是一种比较经典的 BFS 算法，并不是基于 queue 而是基于 known set。即一开始除了出发点外都是未知点，从出发点开始，更新所有与之相邻的点的所求量的值（如最短路径，这里有 greedy 的需求），然后将它的 known 设为 1，找新的扩散源扩散。

与 based on queue 的 BFS 算法不同之处在于，queue-BFS 的新的扩散源是 dequeue 的元素，而 set-BFS 的扩散源需要根据题目要求进行选择，从而满足 greedy 的需求。所以 set-BFS 并不是层层扩散，新的扩散源可能是任何一个 known-set 中的点。由于选择新的扩散点的过程有时间消耗，所以 set-BFS 的复杂度通常更高，但是它能解决 queue-BFS 解决不了的问题：那就是 queue 必须要求新的扩散源扩散出来的点不会影响队列中剩余点作为扩散源的扩散，而 set-BFS 无此要求。

很抽象，所以拿经典的不带权最短路和带权最短路算法（Dijkstra）举例。分析一下两个问题的区别，本质区别在于 greedy 的强弱有区别，不带权时，假如确定了某一层某些点的最短距离（所谓同层是指到起点经过最短边数相同的点），那么与之相邻的下一层的点的最短距离就是再加上 1，而不可能通过还未遍历到的层再到达它取得最短路，所以层推进，即用 queue，是合理的。而带权时，可能通过其它层到达使得其最短，所以必须采用 set 推进。



7. 负权最短路径算法

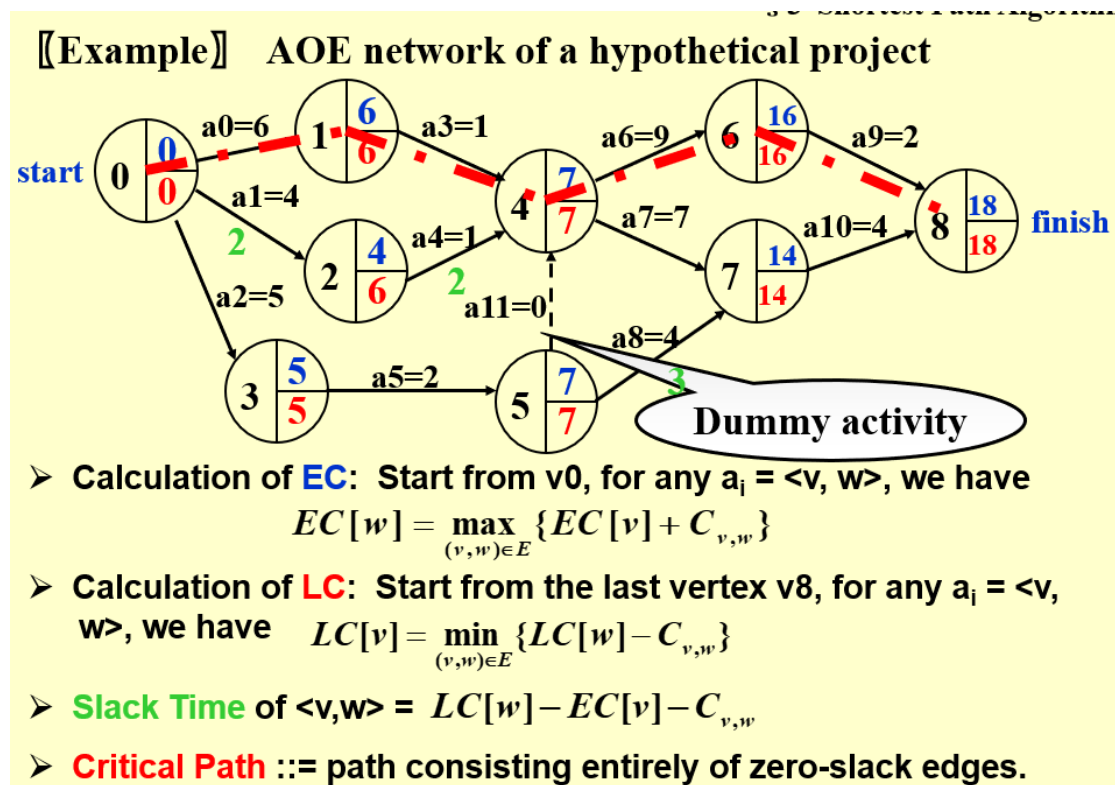
```

void WeightedNegative( Table T )
{ /* T is initialized by Figure 9.30 on p.303 */     $T = O(|V| \times |E|)$ 
  Queue Q;
  Vertex V, W;
  Q = CreateQueue (NumVertex ); MakeEmpty( Q );
  Enqueue( S, Q ); /* Enqueue the source vertex */
  while ( !IsEmpty( Q ) ) { /* each vertex can dequeue at most |V|
    V = Dequeue( Q );    times */
    for ( each W adjacent to V )
      if ( T[ V ].Dist + Cvw < T[ W ].Dist ) { /* no longer once
        T[ W ].Dist = T[ V ].Dist + Cvw;    per edge */
        T[ W ].Path = V;
        if ( W is not already in Q )
          Enqueue( W, Q );
        } /* end-if update */
    } /* end-while */
  DisposeQueue( Q ); /* free memory */
} /* negative-cost cycle will cause indefinite loop */

```

有负权的情况更加复杂，这个算法准确来说不能算 BFS，因为不到最后没有任何一个点的最短路径是已经被确定了的，因为带负权不仅可以跨层，还可以穿过已经被更新距离的点（可能穿过的负权多，穿过后反而路径更短了，这与非负权的 greedy 性质有本质区别）。这里用 queue 和 set 都可以，复杂度是很高的。

8. 图 BFS (前后两次)-AOE 网算法



这个代码考的比较少，主要还是理解 BFS 的思想，它先用拓补排序确定了各层，然后用上面的 EC 公式一层一层地更新 EC 值。最后 EC 值更新完毕后，从终点开始，反向一层一层地更新 LC，最后分析 Slack Time 即可。

主要思路是利用拓补排序得到的层，去得到每个点的某个量。这个量，只跟上一层的量，和上一层与这个点的层之间的连接所决定。这个量在这个问题里是 EC，在不带权最短路径中是最短距离。这种方法类似于对层做数学归纳法，拓补排序是基础。

总结一下 BFS 的三种情况，对于某个层中的点的关键量：

关键量只跟上一层和上一层与这一层的连接边所决定，用 queue-BFS，拓补排序是基础。

关键量可能跟上一层和往后若干层决定，不会跟已经确定的层（上一层再之前的层）有关，用 set-BFS。

关键量可能还与已经确定的层（上一层之前的层）有关，甚至可能根本无法完全确定关键量，queue 强行遍历，复杂度高，严格来说不属于 BFS。

9. 最大流算法

属于特殊技巧型算法，分为如下几步：

复制原图作为残差图 Gr

找一条 Gr 中从起点到终点的路 L (任何一条路径，可用不带权最短路算法，也可以找特定的路径来优化，略)

找到这条路上权最小的边的权 w。对 Gr 中 L 的每条边 (u, v)，做如下变换 (f 代表边的权重)：

$f(u, v) -= w$; //删流

$f(v, u) += w$; //unable undo, 允许回流/撤销操作

$maxflow += w$; (maxFlow 是全局变量，最开始设为 0)

(4) 循环 2~3，直到 Gr 中找不到起点到终点的路径。

(5) 返回 maxFlow 作为最大流。

具体原理跟问题本身的性质有关，可以去证明一下。

10. 图 BFS-Prim 算法

Prim 算法求最小生成树算法与 Dijkstra 算法极其相似，基本一样，都是基于 set-BFS 的算法，与 Dijkstra 唯一不同的地方在于，关键量不是最短距离，而是 lowCost，指的是：将这个点加入生成树所需要增加的最小权重，并且一旦加入 known-set 就失去了意义。Prim 比 Dijkstra 更简单，因为新点的关键量跟之前已经确定的关键量没关系，只跟 known-set 与这个点的连接有关，并且就等于这些连接边的最小权重 (但是复杂度跟 Dijkstra 一样)。

另外一点就是 Prim 目标不是记录终点的关键量，而是在新点入 set 的时候把那个最小权加到全局变量 minWeight 上，记录最小生成树的总权。

11. Kruskal 算法

Kruskal's Algorithm – maintain a forest

```
void Kruskal ( Graph G ) T = O( |E| log |E| )
{ T = { };
  while ( T contains less than |V| -1 edges
    && E is not empty ) {
    choose a least cost edge (v, w) from E ; /* DeleteMin */
    delete (v, w) from E ;
    if ( (v, w) does not create a cycle in T )
      add (v, w) to T ; /* Union / Find */
    else
      discard (v, w) ;
  }
  if ( T contains fewer than |V| -1 edges )
    Error ( "No spanning tree" ) ;
}
```

这是一种基于并查集的算法，Prim 是加点，Kruskal 是加边，加的边满足：不会形成圈的、权重最小的边。可以证明这种 greedy 成立。验证不构成圈的方法是并查集：并查集中，两个点在同一集合里的条件是：当前已经选的边让这两个点互相连通。新加的边的两个点，不能属于同一集合，加完后，再把这两个点 union 即可。

12. 图 DFS-生成森林算法

课程里没有重点讲 DFS 生成森林，只有一页 ppt，但我觉得这是 DFS 最重要的点，后面基于 DFS 的算法都是生成森林的延伸和应用，所以单独写了一节。

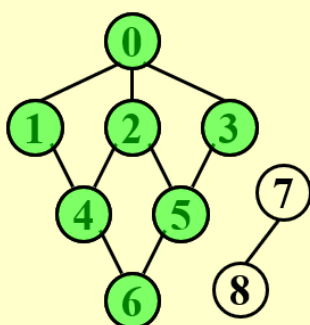
§ 6 Applications of Depth-First Search

/ a generalization of preorder traversal */*

```
void DFS ( Vertex V ) /* this is only a template */
{ visited[ V ] = true; /* mark this vertex to avoid cycles */
  for ( each W adjacent to V )
    if ( !visited[ W ] )
      DFS( W );
} /* T = O( |E| + |V| ) as long as adjacency lists are used */
```

1. Undirected Graphs

DFS (0)



```
void ListComponents ( Graph G )
{ for ( each V in G )
  { if ( !visited[ V ] ) {
    DFS( V );
    printf( "\n" );
  }
}
0 1 4 6 5 2 3
7 8
```

这一页信息量不小，给出了 DFS 的伪代码，关键点在于，这个算法能生成一个图的生成森林。所谓生成森林，其实就是生成若干个树来覆盖图的每个顶点，从而实现 $O(V+E)$ 的遍历，并且每次都是沿着一条路径遍历到底，然后返回再探其他最深路，这与 BFS 遍历的扩散机制有根本区别。

对于森林中的每棵树，DFS 的遍历顺序是前序遍历（如果假定相邻的节点是从左到右 DFS）。这里的遍历指的是访问到该节点的下标，并不意味着输出，或者

操作。也就是说 DFS 实现后序遍历操作是完全可以的，具体方法是：

在 DFS 函数最开始进行操作（如输出节点下标）：前序遍历

在 DFS 函数返回前加操作语句：后序遍历

```
void DFS ( Vertex V ) /* this is only a template */
{ visited[ V ] = true; /* mark this vertex to avoid cycles */
  for ( each W adjacent to V )
    if ( !visited[ W ] )
      DFS( W );
} /* T = O( |E| + |V| ) as long as adjacency lists are used */
```

前序
后序

前序遍历后序遍历分别是访问时操作和回溯时操作。

注意图是生成树很可能不是二叉树，所以中序遍历是没有什么意义的。

DFS 生成森林的树的个数也可以统计，上面那 ppt 下面的生成森林算法中输出换行符的位置就标志着一个生成树的结束。

13. 图 DFS-强连通分量算法

强连通分量是指有向图的点的子集，里面点两两互相连通。这个算法没有在课堂上讲，书里有，作业编程题有，应该不会考，但可以帮理解 DFS。分为三步。

- (1) 第一次 DFS，给点编号，编号顺序为生成森林的后序遍历。
- (2) 将图反转（每条边反向）成 G_r 。
- (3) 第二次 DFS，编号大的点优先，同时输出各个树分量。

精髓在于，在给点编号、大编号优先的情况下第二次 DFS，得到的森林的各个树就是强连通分量。

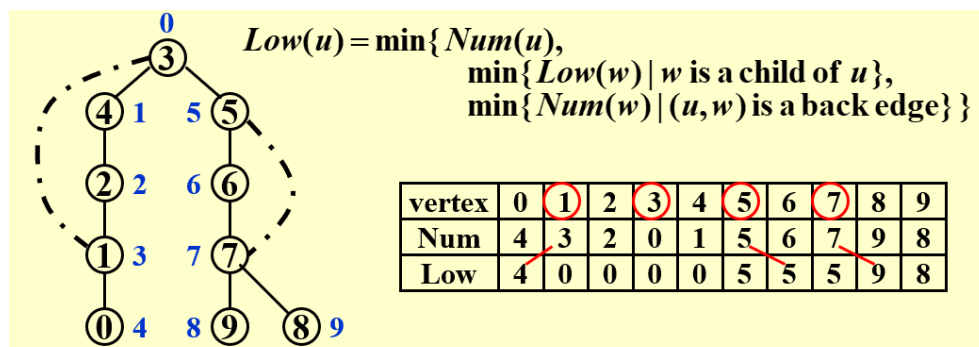
具体原理证明略，可以思考一下。

14. 图 DFS-关键节点算法

关键节点是删除它就会导致本来连通的图不再连通的节点，关键节点数反应了图的连接性的可靠性，此算法要求找出所有关键节点。属于技巧型算法，分为如下几步

- (1) 随意选取一个根节点，生成覆盖整个图的树（生成不了说明图的连通性前提不成立），按照前序遍历顺序给顶点编号。这意味着祖先的编号一定比孩子要小。注意在生成树的同时，要额外添加 back edge，也就是在原图里但是不在树里的边，这里有一个几乎不消耗复杂度的方法就是在 ppt 中的 DFS 函数里的 `if(!visit[W])` 这个条件语句后加一条 `else` 语句，里面的内容就是添加 $V \rightarrow W$ 的 back edge。注意要给 back edge 加一个标志区别于树的边。注意在生成树的同时要记录一下各节点的 Parent，后面会用。

(2) 按如下原则，构造一个 Low 数组



Therefore, u is an **articulation point** iff

- (1) u is the **root** and has **at least 2 children**; or
- (2) u is not the root, and has **at least 1 child** such that $Low(child) \geq Num(u)$.

具体方法是写一个递归的 AssignLow 函数，书里有，本质上就是从下往上推出 Low。

(3) 最后就是根据 ppt 的原则依次判定每个点是否是关键节点。

```

/* Assign Num and compute Parents */
void
AssignNum( Vertex V )
{
    Vertex W;
    Num[ V ] = Counter++;
    Visited[ V ] = True;
    for each W adjacent to V
    {
        if( !Visited[ W ] )
        {
            Parent[ W ] = V;
            AssignNum( W );
        }
    }
}

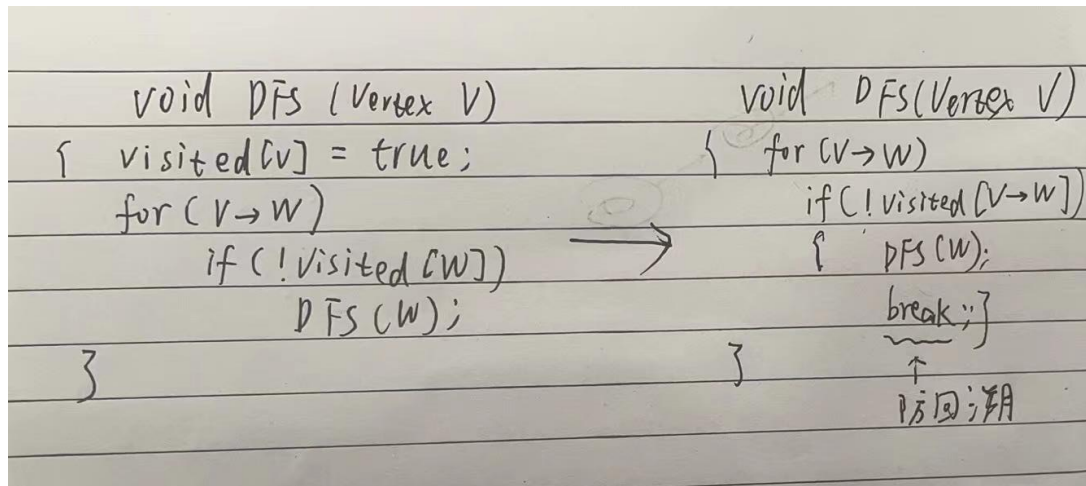
/* Assign Low; also check for articulation points */
void
AssignLow( Vertex V )
{
    Vertex W;
    Low[ V ] = Num[ V ]; /* Rule 1 */
    for each W adjacent to V
    {
        if( Num[ W ] > Num[ V ] ) /* Forward edge */
        {
            AssignLow( W );
            if( Low[ W ] >= Num[ V ] )
            {
                printf( "%v is an articulation point\n", v );
                Low[ V ] = Min( Low[ V ], Low[ W ] ); /* Rule 3 */
            }
        }
        else if( Parent[ V ] != W ) /* Back edge */
        {
            Low[ V ] = Min( Low[ V ], Num[ W ] ); /* Rule 2 */
        }
    }
}

```

这种算法的原理跟关键节点的定义密切相关。

16. 图 DFS-欧拉图算法

欧拉圈是指经过图中每个顶点但不过重复边的圈（欧拉路径类似定义）。求欧拉圈是 DFS 的经典应用之一，方法很简单，就是 DFS 生成树，但是不回溯，这样生成的树就退化成了——一条路径（圈）。另外，与普通 DFS 不同的是，它终止继续探寻的条件不是点已经被访问过，而是它的所有伸出的边都已经被走过。所以 DFS 函数有如下改动。



如果没有遍历到所有的点，则说明没有欧拉路径。如果没有遍历到所有的点并回到起点，说明没有欧拉圈。

Appendix. C-defined Queue

由于很多算法都要用到队列，而不确定考试时能否使用 C++，所以给出了 C-defined Circular Queue 的相对好一点的写法。（期中考试就是因为 c 写的 queue 出了一个低级 bug 导致编程题扣了好多分）。

```
typedef struct Queue//C-defined circular queue
{
    int* qArr;//store the element of the queue
    int front;//q->front points to the element that is going to be popped.
    int back;//q->back points to the position where the new element is going
    to enqueue at.
    int QueueLen;//stores the number of elements in queue
    int MaxLen;//the longest length
}Queue;

void ConstructQueue(Queue* q, int maxl)
{
    if(q==NULL)
        return;
    q->QueueLen=0;
    q->MaxLen=maxl;
    q->qArr=(int*)malloc((q->MaxLen)*sizeof(int));//queue is malloced
    q->front=0;//q->front points to the element that is going to be popped.
```

```

        q->back=0;//q->back points to the position where the new element is going
        to enqueue at.
    }
    int qNext(Queue *q, int ind)
    {
        return (ind+1)%q->MaxLen;//the next index in this circular queue
    }
    void Enqueue(Queue* q, int val)
    {
        if(q->QueueLen+1==q->MaxLen)
        {
            printf("Queue is already full!\n");
            return;
        }
        q->qArr[q->back]=val;
        q->back=qNext(q,q->back);
        q->QueueLen++;
    }
    int Dequeue(Queue* q)
    {
        if(q->QueueLen==0)
        {
            printf("Queue is already empty!\n");
            return -1;
        }

        int ret=q->qArr[q->front];
        q->front=qNext(q,q->front);
        q->QueueLen--;
        return ret;
    }
    bool isEmpty(Queue* q)
    {
        return q->QueueLen==0;
    }

```