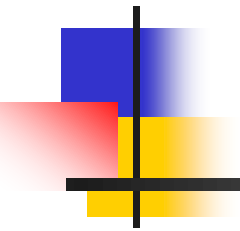


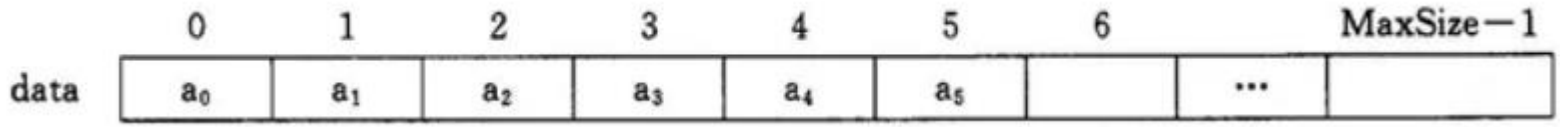
查找与搜索

搜索

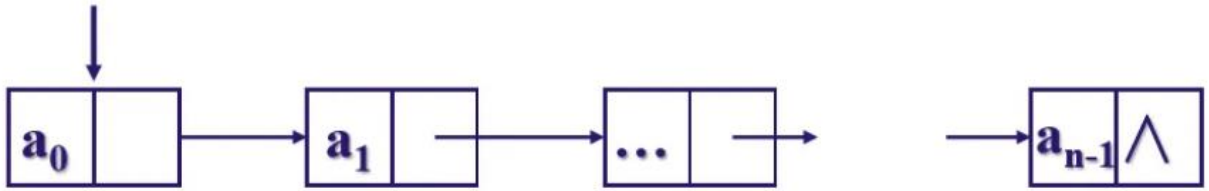
- 数据处理领域最基本的任务之一
  - 查找
  - 检索
    - 查某个指定的值
    - 找最大值
    - 找满足某些条件的值
    - ……
- “搜索就是有计划有规律的枚举”
- 搜索跟数据的组织结构和存储方式有关！
- 搜索同时也是一种问题求解策略和方法！

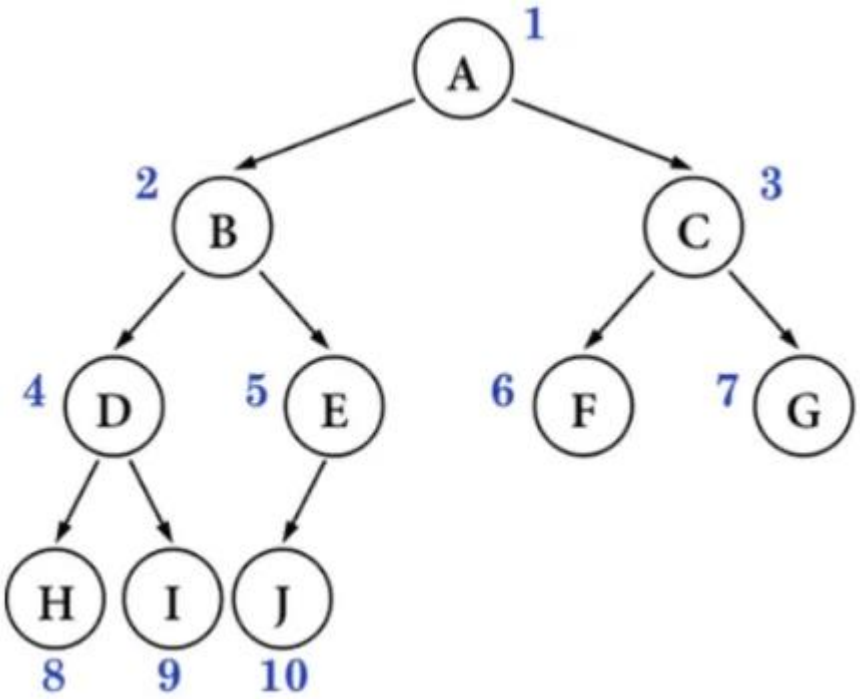
常见的数据结构

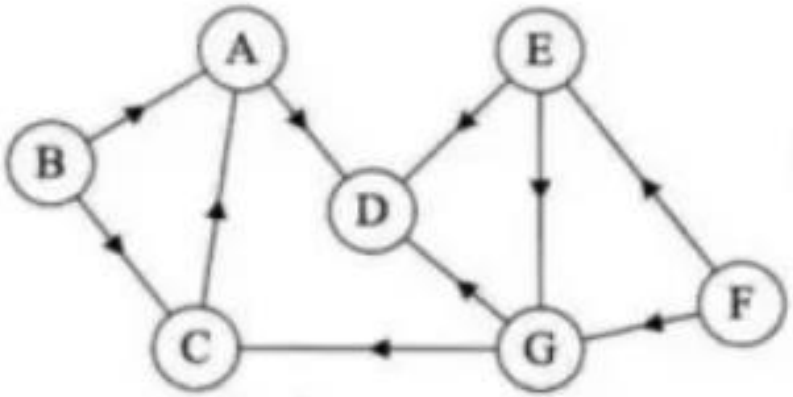
- 线性结构：数组、栈、队列、链表
- 树形结构：树、二叉树，以及各种变型
- 图形结构：各种图
  - 图
  - 搜索


0
1
2
3
4
5
6
MaxSize−1
data
a0
a1
a2
a3
a4
a5
...

head

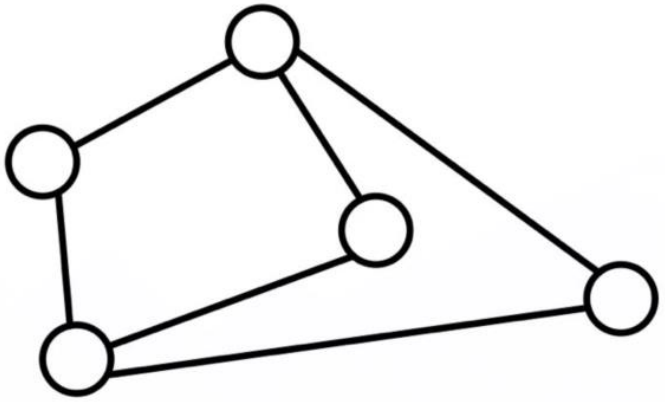

a0
a1
...
an-1
∧


1
A
2
B
3
C
4
D
5
E
6
F
7
G
H
I
J
8
9
10


A
E
B
D
F
C
G

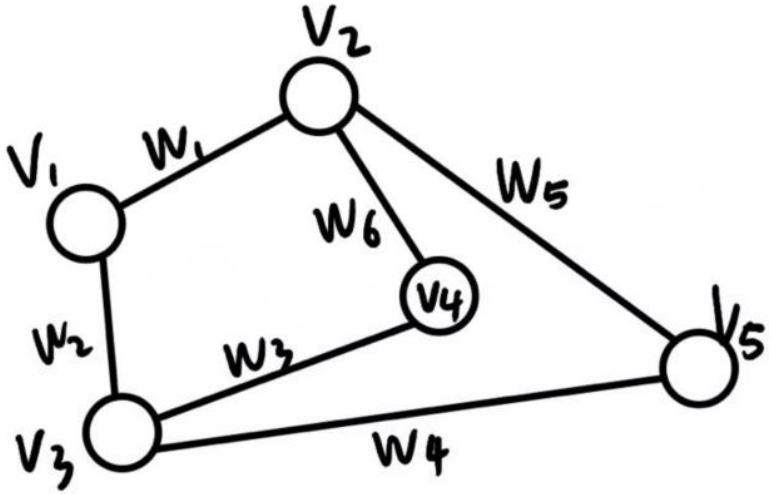
图

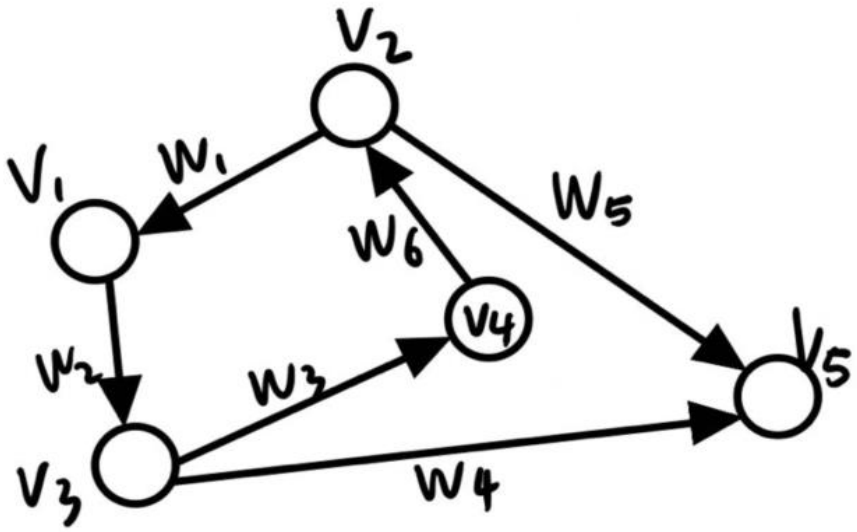
- 图（**Graph**），网络
- 图论（**Graph Theory**）
  数学的一个分支。它以图为研究对象。
- 图论中的图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。
  - 交通路网
  - 人际关系
  - ……



图的基本概念

- 顶点：V1，V2，V3…
- 边：连接两个顶点之间的边
- 边权：w1，w2，w3…
- 有向图：每一条边有箭头的，表示单向关系
- 无向图：每条边都没有箭头，表示双向的关系
- 出度：从顶点**v**出发的边的数目（有向图）
- 入度：指向顶点**v**的边的数目（有向图）

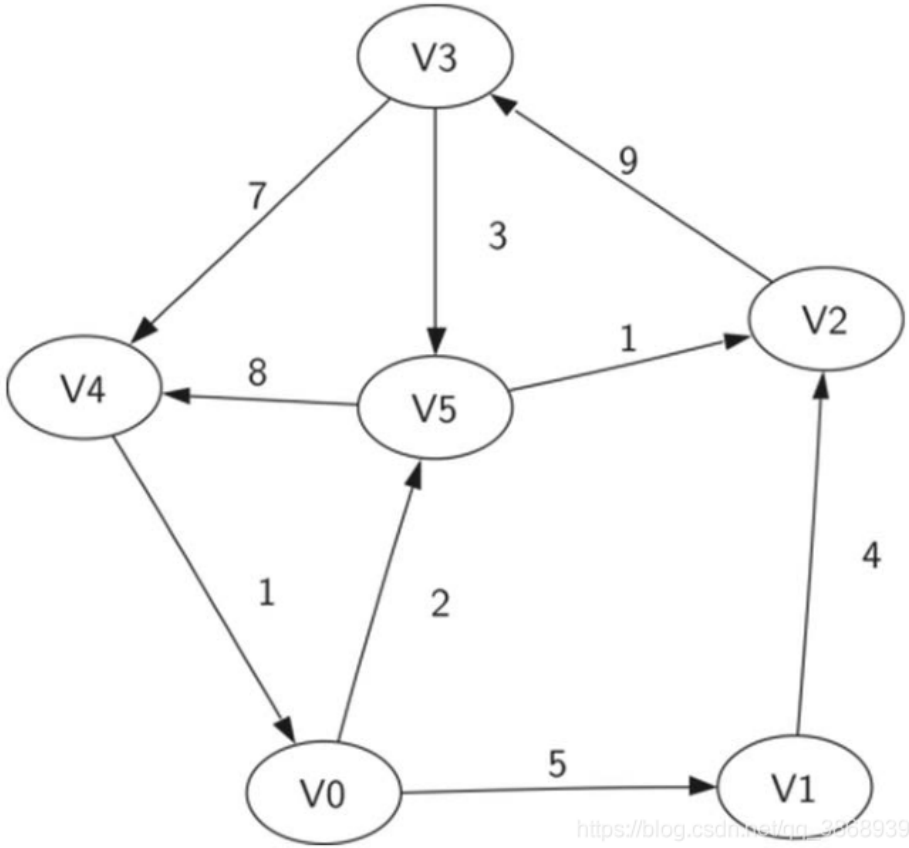


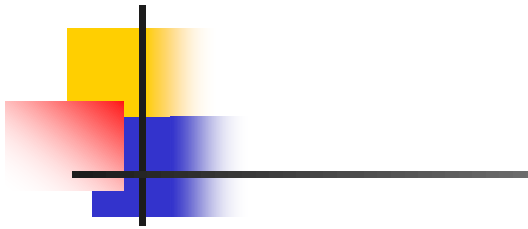


图的存储

- 邻接矩阵

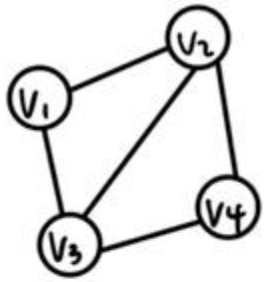
	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

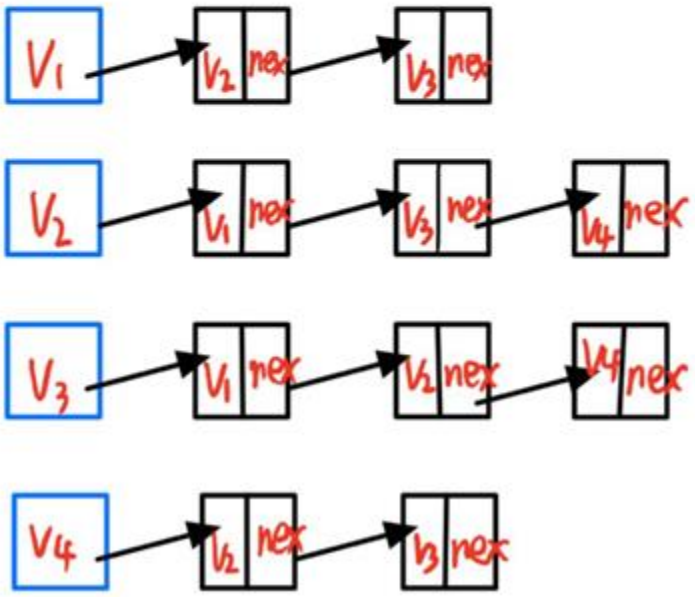




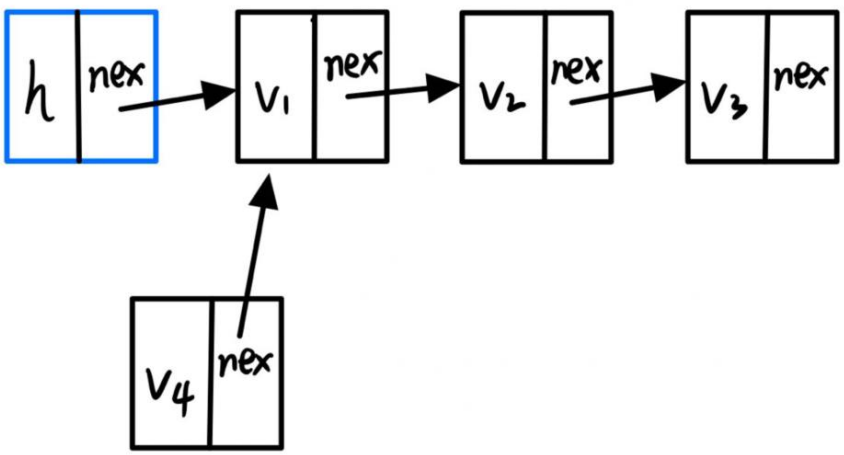
- 二维数组e[i][j]
- 将e[i][j]全部初始化为0（或inf）
- 对于一条边e = (u, v, w)
  - 无向图：e[u][v] = e[v][u] = w
  - 有向图：e[u][v] = w
- 简单、方便
- 空间复杂度是**O(N^2)**，大小受限
- 适合于点少边多的稠密图，否则空间浪费严重
- 不能存重边

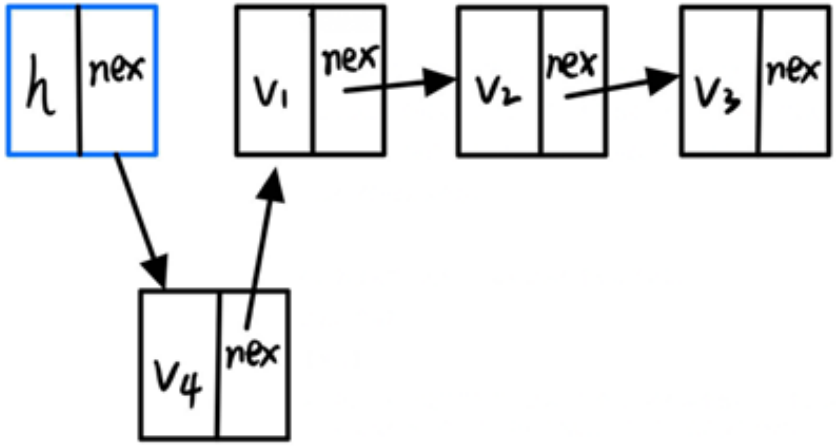
- 邻接表
  - 二维链表；
  - 对每个顶点开一个链表，表头就是该顶点，链表中的元素表示和该点相邻的顶点；
  - 边权存放在链表元素中；
  - 空间复杂度是
    O(N+2M)
    O(N+M)

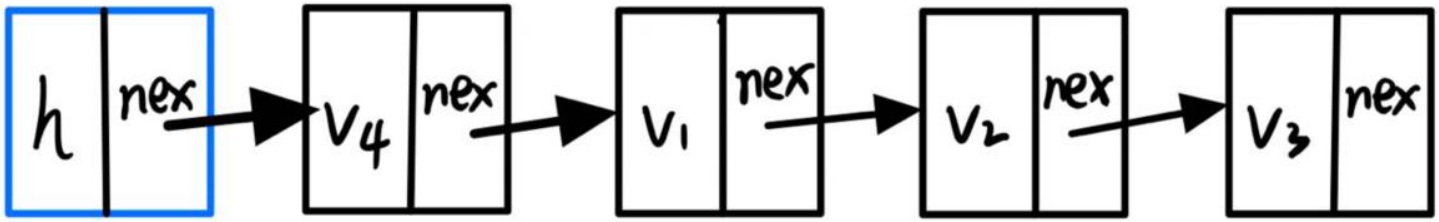




- 如何在链表的头部插入一个新的元素？







- 邻接表的数组实现方式

```
// N为节点数，M为总边数，注意如果是无向图需要开两倍的M
int h[N], to[M], weight[M], nex[M], tot;
void add(int u, int v, int w){
    ++tot;
    to[tot] = v;
    weight[tot] = w;
    nex[tot] = h[u];
    h[u] = tot;
}
```

tot：　当前的总边数，每条边都赋予一个**[1,tot]**的不同的编号
h：　存放顶点，**h[i]**第**i**个顶点的表头；
to：　存放边，每条边都有唯一的编号；
weight：存放边的权重；
nex：　编号为**i**的边的下一条边，
　　相当于链表中的**nex**指针

- 邻接表的C++超级数组实现

```cpp
#include <vector>
using namespace std;
const int maxn = 1e3 + 5;
int main() {
    //点和边的存储结构
    vector< pair<int, int> > vec[maxn];
    //建图
    int n, m;
    scanf("%d%d", &n, &m);
    int u, v, w;
    while (m--) {
        scanf("%d%d%d", &u, &v, &w);
        vec[u].push_back(make_pair(v, w));
        vec[v].push_back(make_pair(u, w));
    }
    //访问图（输出边）
    for (int i = 0; i < n; ++i) {
        for (auto &v : vec[i]) {
            printf("#%d --> #%d : %d\n", i, v.first, v.second);
        }
    }
}
```

图的搜索（遍历）

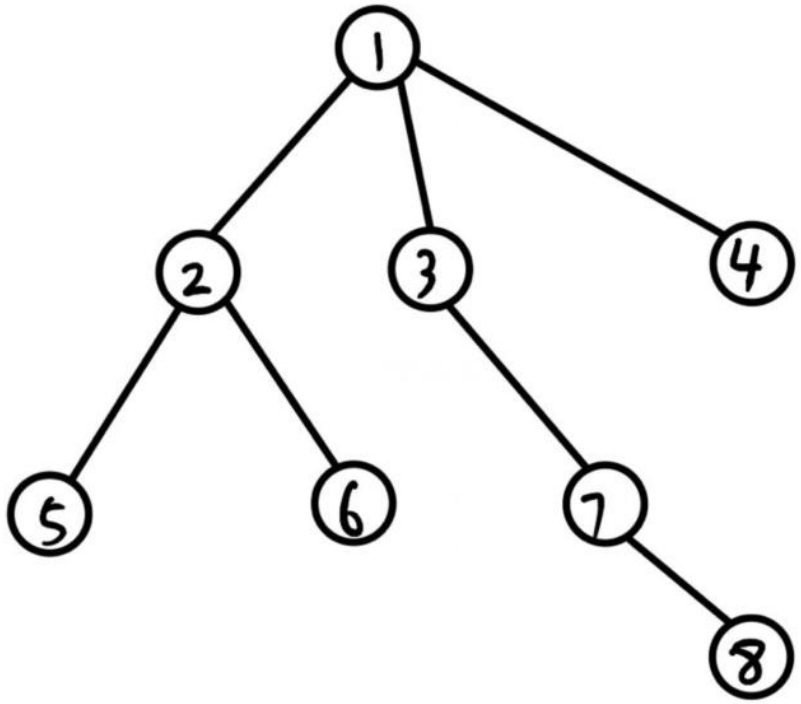
- 深度优先搜索

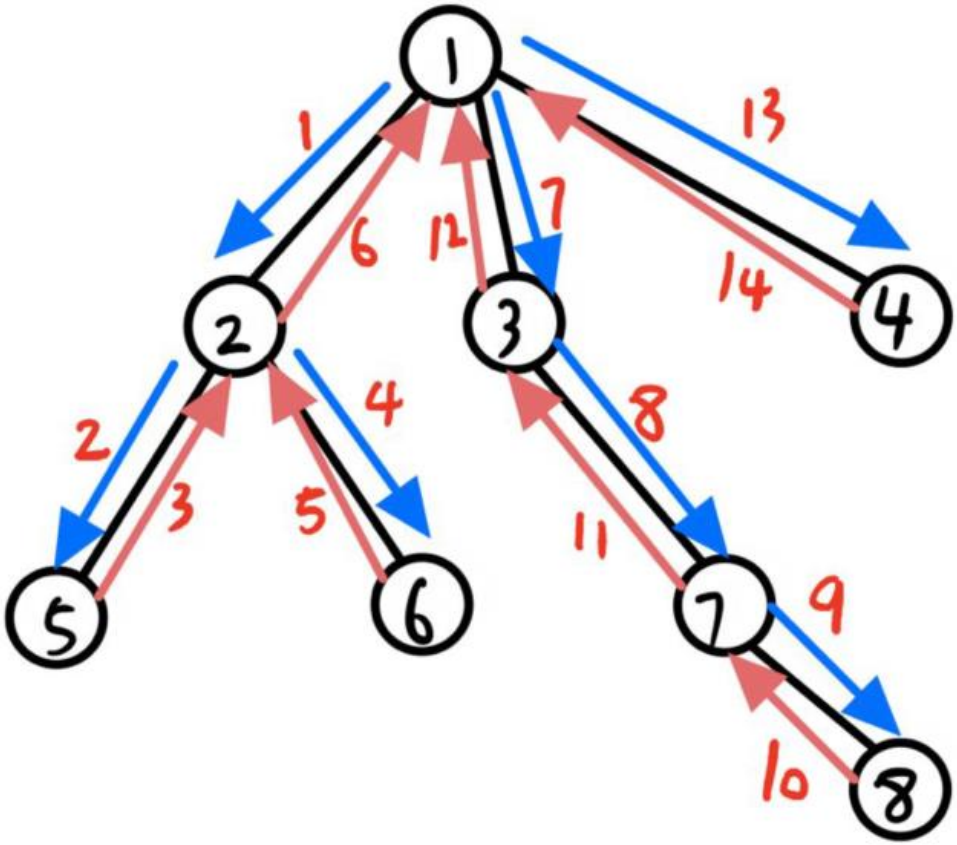
  DFS (Depth-First-Search)

- 广度优先搜索（宽度优先搜索）

  BFS (Breadth-First-Search)

- 树的**DFS**



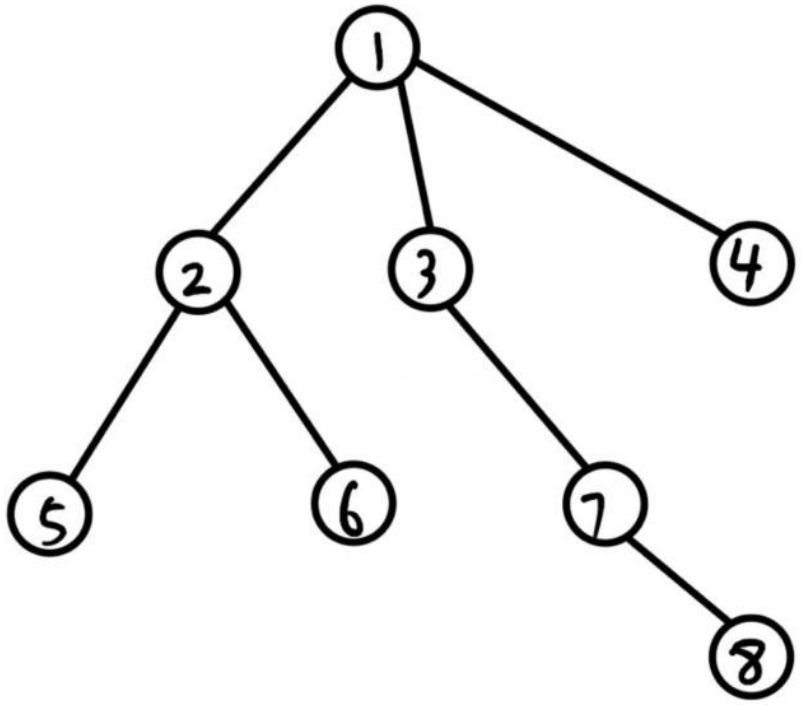


DFS访问结果：1 2 5 6 3 7 8 4

• 树的BFS

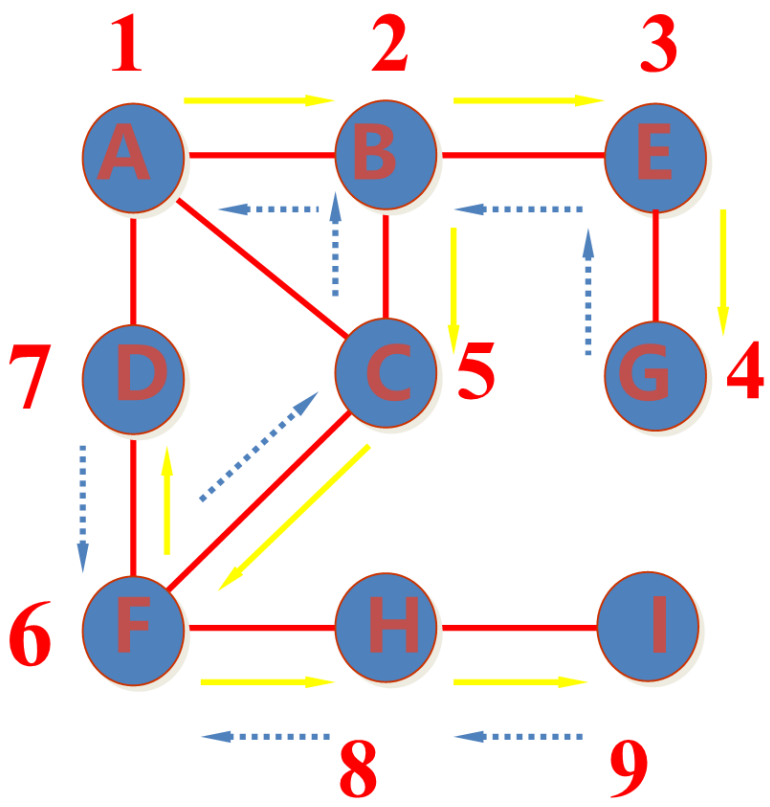
设队列q，一开始为空q = {}

① 1号点入队，q={1}
② 取出队首，front=1，q={}，检查1能够到达的点有2,3,4；将其加入q，q={2,3,4}
③ 取出队首，front=2，检查2能够到达的点有5,6；将其加入q，q={3,4,5,6}
④ 取出队首，front=3，检查3能够到达的点有7；将其加入q，q={4,5,6,7}
⑤ 取出队首，front=4，检查4能够到达的点，无；跳过，q={5,6,7}
⑥ 取出队首，front=5，检查5能够到达的点，无；跳过，q={6,7}
⑦ 取出队首，front=6，检查6能够到达的点，无；跳过，q={7}
⑧ 取出队首，front=7，检查7能够到达的点，8；将其加入q，q={8}
⑨ 取出队首，front=8，检查8能够到达的点，无；跳过
⑩ 结束，遍历顺序为1,2,3,4,5,6,7,8

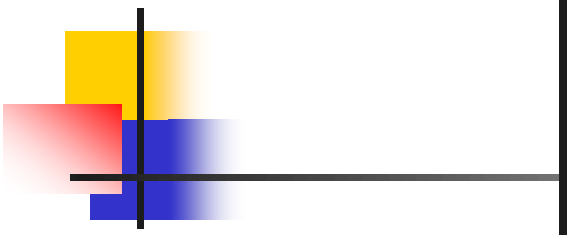


BFS访问结果：1 2 3 4 5 6 7 8

图的DFS



- 先选择一条路径一直访问看能否到达目标，如果不能则回溯到上一节点继续进行搜索，如果到达则根据需要退出或继续遍历整个图。
- 访问过的点需要标记
- 走不下去了就回退；
- 有多条路的时候，可以顺序选择或随机选
- 遍历结果是一棵树。



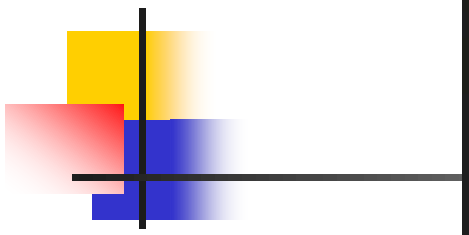
图的**DFS**
搜索框架

到底怎么
走？
记录路径

```
#include <stdio.h>
//Dfs用于判断从v出发能否走到终点
bool dfs(V)
{
    if (V为终点)
        return true;
    if (V为旧点)
        return false;
    将V标记为旧点;
    对和V相邻的每个节点U
    {
        if (dfs(U)==true)
            return true;
    }
    return false;
}
```

```
NODE path[节点总数];
int depth;
//判断从V出发能否走到终点，
//如果能，要记录路径
bool dfs(V)
{
    if (V为终点)
    {
        path[depth]=V;
        return true;
    }
    if (V为旧点)
        return false;
    将V标记为旧点;
    path[depth]=V;
    depth++;
    对和V相邻的每个节点U
    {
        if (dfs(U)=true)
            return true;
    }
    depth--;
    return false;
}
```

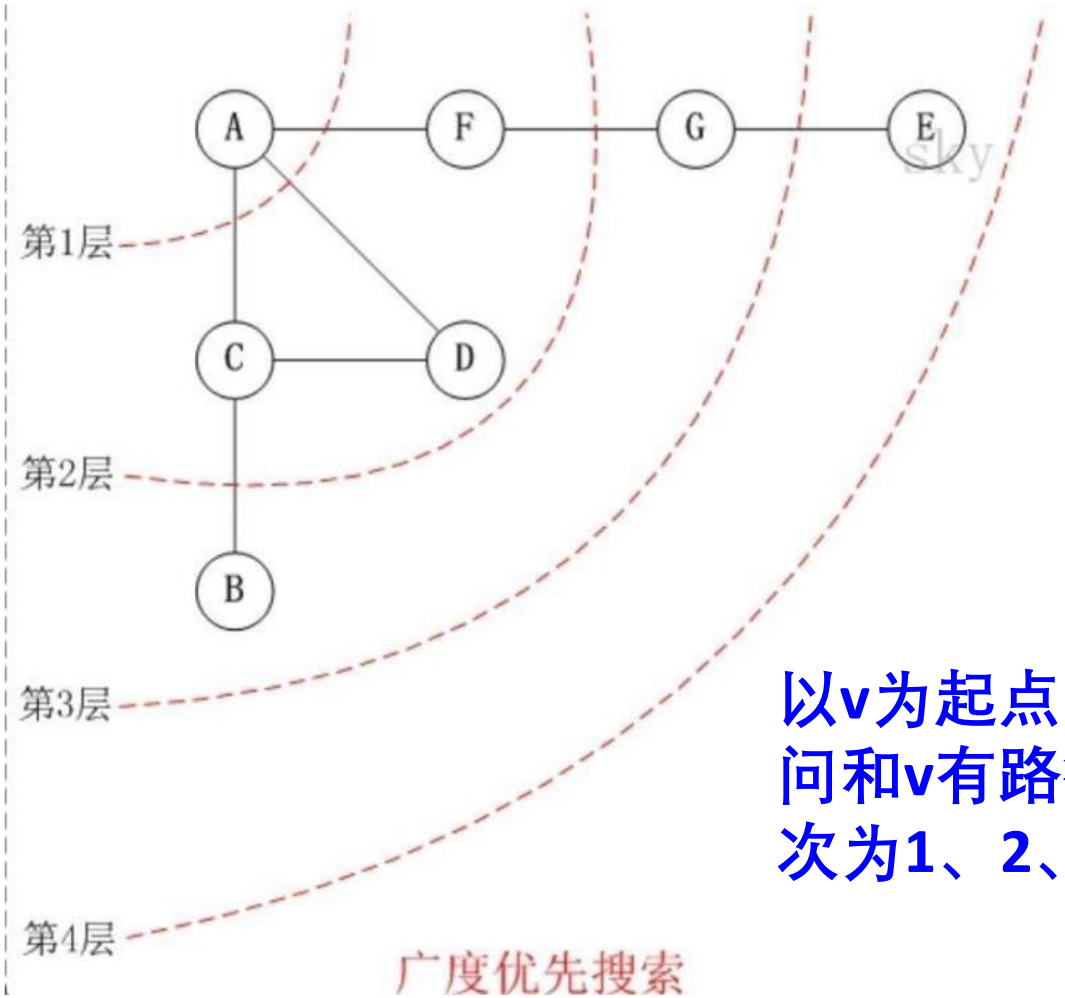
```
if (dfs(起点))
{
    for (i=0;i<=depth;i++)
    {
        printf ("%d",path[i]);
    }
}
```



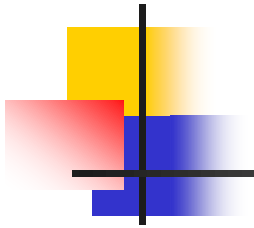
图的DFS遍历

```
Dfs(V)
{
    if(V是旧点)
        return;
    将V标记为旧点;
    对和V相邻的每个点U
    {
        Dfs(U);
    }
}
int main(void)
{
    将所有点都标记为新点;
    while(在图中可以找到的新点K)
        Dfs(K);
}
```

- 图的**BFS**

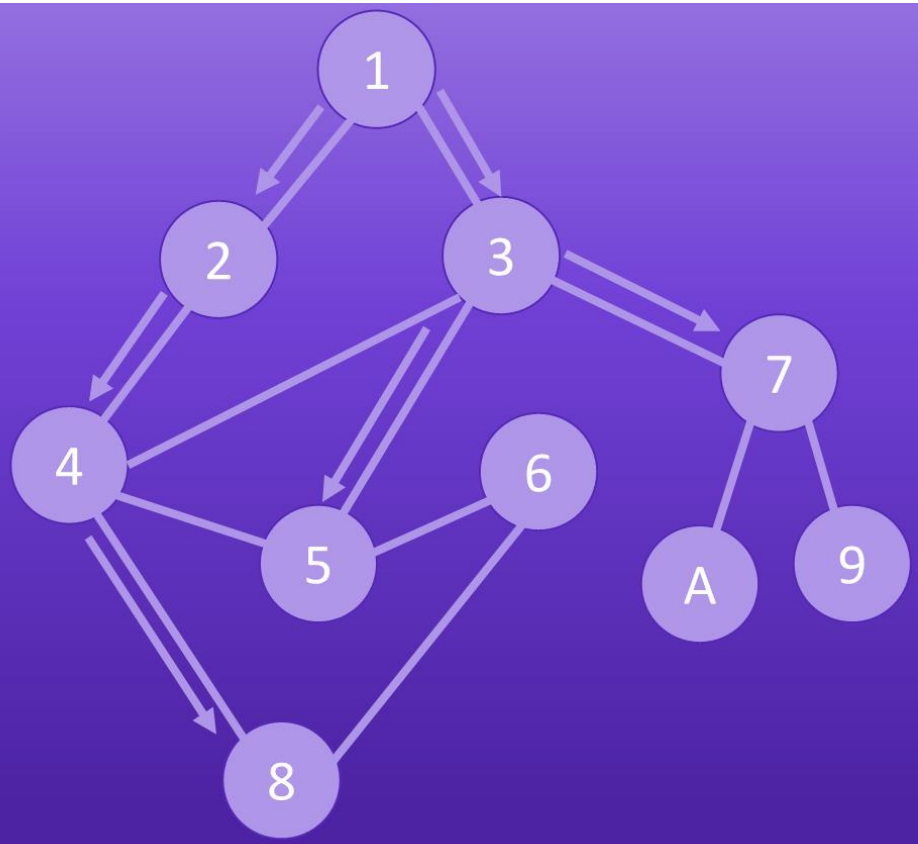


以v为起点，由近至远，依次访问和v有路径相通且路径长度依次为1、2、…，的顶点



- 给节点分层；起点是第**0**层；从起点最少需**n**步就能到达的点属于第**n**层；
- 依层次顺序，从小到大扩展（访问）节点；把层次低的点全部扩展（访问）出来后，才会扩展（访问）层次高的点；
- 记录每个点被访问的次序及步数
  队列
- 记录点是否被访问过
  数组

- 将起点1放入队列；
- 从队列中取出起点1进行访问，将与1相邻的未访问顶点2、3放入队列；
- 从队列开头取出2进行访问，将与2相邻的未访问顶点4放入队列；
- 从队列开头取出3进行访问，将与3相邻的未访问顶点5和7放入队列；
- 从队列开头取出4进行访问，将与4相邻的未访问顶点8放入队列；



图的BFS搜索

```
// BFS搜索
queue<int> q;
bool vis[N]={false};//记录点的访问情况
int bfs( int s,int t){
    if (s==t)
        return s;//找到
    vis[s]=true;
    q.push(s);
    while(!q.empty()){
        int cur=q.top(); q.pop();
        //检查当前结点的所有边
        for(int e=h[cur];e;e=nex[e]){
            int v=to[e]; //边关联的点
            if(!vis[v]){
                if (v==t)
                    return v;//找到
                vis[v]=true;
                q.push(v);
            }
        }
    }
    return -1;//未找到
}
```

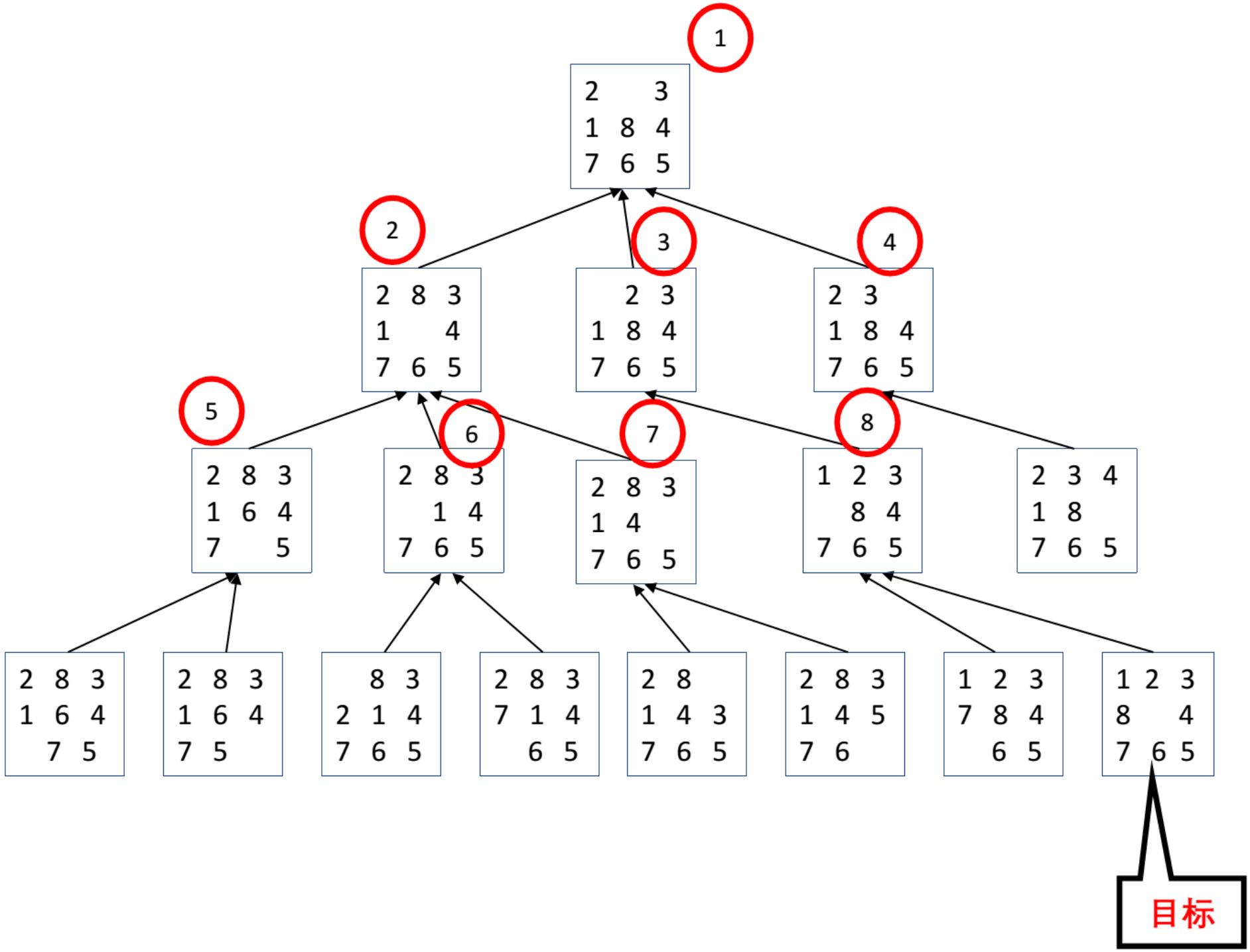
图的BFS遍历

```
// BFS 遍历
queue<int> q;
bool vis[N]={false};//记录点的访问情况
void bfs( int start){
    printf("%d",cur); //访问结点
    vis[start]=true;
    q.push(start);
    while(!q.empty()){
        int cur=q.top(); q.pop();
        //检查当前结点的所有边
        for(int e=h[cur];e;e=nex[e]){
            int v=to[e]; //边关联的点
            if(!vis[v]){
                printf("%d",cur); //访问结点
                vis[v]=true;
                q.push(v);
            }
        }
    }
}
```

搜索应用：八数码问题

- 在3× 3的棋盘，摆有八个棋子，每个棋子上标有1至8的某一数字，不同棋子上标的数字不相同。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。
- 给出一个初始状态和一个目标状态，找出一种从初始转变成目标状态的移动棋子步数最少的移动步骤。

1	2	3
8		4
7	6	5


1
2 3
1 8 4
7 6 5
2
2 8 3
1 4
7 6 5
3
2 3
1 8 4
7 6 5
4
2 3
1 8 4
7 6 5
5
2 8 3
1 6 4
7 5
6
2 8 3
1 4
7 6 5
7
2 8 3
1 4
7 6 5
8
1 2 3
8 4
7 6 5
2 3 4
1 8
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 6 4
7 5
8 3
2 1 4
7 6 5
2 8 3
7 1 4
6 5
2 8
1 4 3
7 6 5
2 8 3
1 4 5
7 6
1 2 3
7 8 4
6 5
1 2 3
8 4
7 6 5
目标

- 如何判重
  - 每个状态可转化成唯一的值，然后再采用**vis**布尔数组存储的方式判断是否重复到达。
  - 更好的办法？

1	2	3
8		4
7	6	5

123804765 (或 12380476)

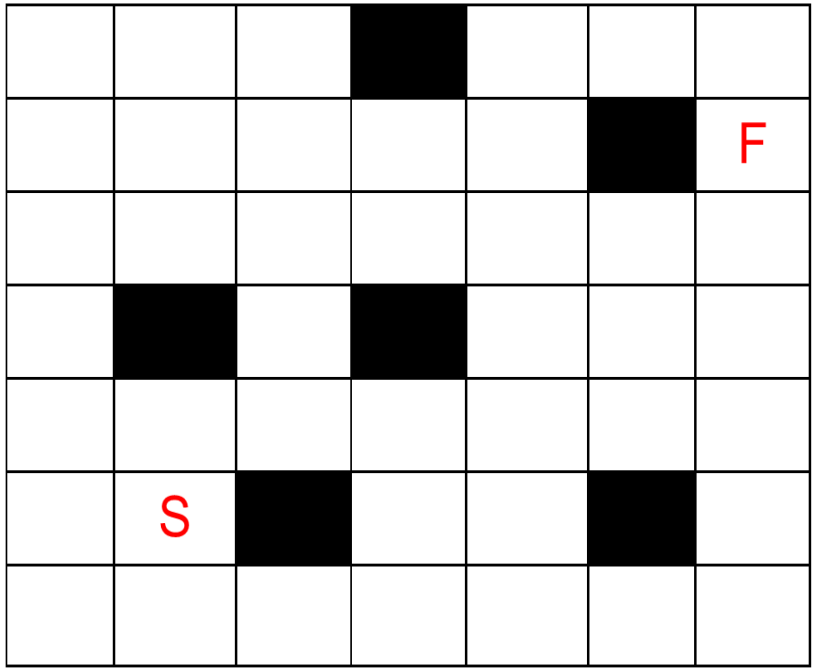
搜索应用：走迷宫问题

- 给定01矩阵地图， 0表示地面， 1表示障碍物， 可以上下左右移动。

（1）给定起点终点，求不交叉路径数；

（2）给定起点终点，求最少移动步数（最短路线）

00001
10101
10100
00000



._.	0	1	2	3	4	5	6
0	6	7	6		8	9	
1	5	6	5	6	7		F
2	4	5	4	5	6	7	8
3	3		3		5	6	7
4	2	1	2	3	4	5	6
5	1	S		4	5		7
6	2	1	2	3	4	5	6

搜索应用：跳方格

- 有一个房间，地板由黑色或者红色的方块铺成的。一个人站在一块黑色的方块上，他可以上下左右移动，但是他只能走黑色的方块，问他能到的方块一共有多少块。
- **BFS**每到达一个点就**ans++**

搜索应用：有障碍的迷宫

- 天使被困在监狱，他的朋友们想见他，监狱的地形复杂，包括路(用点标示)，墙(用#标示)，天使的位置(用a标示)，他的朋友(用r标示)，监狱里还有守卫(用x标示)，他的朋友只能向左右上下四个方向走动，每走一步花一单位时间，若碰上守卫，消灭守卫需要额外花费一单位时间。问最少多长时间天使能见到他的朋友。
- 原题：多个起点，一个终点；起终点互换，变为：一个起点，多个终点；
- 使用**BFS**，遇到**x**时**step+1** ，位置不变放入队列。

搜索应用：电梯问题

- 有一特别的电梯，第 **i** 层有一个对应的数字 **ki**，对于第 **i** 层按上升键 **up** 可升上到 **i+k[i]** 层，按下降键 **down** 到达 **i-k[i]** 层，到达的楼层最高不能超过 **n** 层，最低不能小于 **1** 层。给你一个起点 **A** 和终点 **B**，问最少要按几次上升键或者下降键到达目的地。若无法到达，则输出 **-1**。
- **BFS**
  - 从起点开始按要求搜索
  - 每层有上和下两个扩展
  - 注意边界的扩展

搜索应用：N皇后问题

- 在N*N的方格棋盘放置了N个皇后，使得它们不相互攻击（即任意2个皇后不允许处在同一排，同一列，也不允许处在45度对角线上。对于给定的N，求出有多少种合法的放置方法。
- 对行（或者列）进行深度优先搜索：
  - 对每一行，枚举可放在哪一列，判断和之前行有没有冲突，同时检查跟对角线上的是否冲突；
    - 递归：从第一行开始，不断地递归处理下一行，直到处理到n+1 行就停止。
    - 回溯：若该行的任意位置均不能满足条件，则进行回退，退到上一行的下一个状态。

搜索优化——剪枝

- 剪枝，一种搜索优化手段，目的是排除掉一些显然不会满足题目要求的情况，减少搜索次数。
- 剪枝可分为
  - 最优性剪枝——当前的搜索结果一定不是最优解了，就不再搜索下去，回溯搜索其他情况；
  - 可行性剪枝——当前的搜索结果不可取，连题目要求都达不到，就不再搜索下去，回溯搜索其他情况。

搜索剪枝：素数环

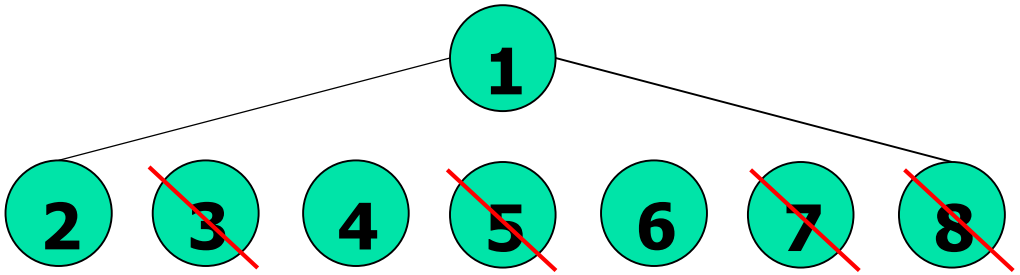
- 从1到n(n<=20), n个数摆成一个环，要求相邻的两个数的和是一个素数，共有多少种摆法？

  n=8时，有4种

  1 2 3 8 5 6 7 4
  1 2 5 8 3 4 7 6
  1 4 7 6 5 8 3 2
  1 6 7 4 3 8 5 2

- 基本思路：采用**DFS**，第一个位置始终填**1**，从第二个位置开始，每个位置都尝试填其余的 **n-1** 个数字，利用递归对所有情况进行遍历。

- 当n为奇数时，一定无解。因为n为奇数，则必有两个奇数会相邻，其和不是素数（除了2）；
- 当n为偶数时
  - 剪枝优化：利用相邻两数之和为素数进行剪枝，在确定当前位置所填数的情况下，不需要搜索那些不满足素数和要求的分支；
  - 深度达到n时，不用再扩展，判断与第一个位置的和是否为素数即可。



搜索剪枝：Roads问题

- 有N个城市(编号1到N)。城市间有R条单向道路。每条道路连接两个城市，有长度和过路费两个属性。 某人只有K元钱，他想从城市1走到城市N，最短共需要走多长的路？如果到不了，输出-1。
- 基本思路：
  从城市1开始，DFS遍历整个图，找到所有能够到达N的走法，选一个最优的。

- 剪枝优化：
  - 记录当前已知的从起点到达某个城市i的最短路径长度L，如果在搜索过程中，遇到某条到达城市i的路径长度已经大于等于L，可以直接放弃，不用走到底了。
  - 如果花了同样的钱，走到同一城市，但是走的路比以前花同样的钱走的路短或等于，这条路就需要停止了。
  - 如果当前路径到达下一个城市的路费超过总费用，也不用再走了。

- 加上剪枝的DFS框架

```
void dfs(状态,深度)
{
    if(终止条件)
    {
        一些处理…
    }

    if(剪枝条件)
    {
        一些处理…
        return;
    }

    for(枚举可行下一个状态)
    {
        状态=下一个状态;
        dfs(状态,深度+1);
        状态=原状态 ;
    }
}
```

DFS和BFS的比较

- **BFS**一般适用于状态表示比较简单、求最优策略的问题。
  - 优点：是一种完备策略，即只要问题有解，它就一定可以找到解， 并且找到的一定是路径最短的解。
  - 缺点： 盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用的节点， 因此其搜索效率较低。需要保存所有扩展出的状态，占用的空间大。
- **DFS**几乎可以用于任何问题， 只需要保存从起始状态到当前状态路径上的节点。