

## 树与图的存储

树是一种特殊的图，与图的存储方式相同。

对于无向图中的边 $ab$ ，存储两条有向边 $a \rightarrow b, b \rightarrow a$ 。

因此我们可以只考虑有向图的存储。

(1) 邻接矩阵:  $g[a][b]$  存储边  $a \rightarrow b$

(2) 邻接表:

```
1 // 对于每个点k, 开一个单链表, 存储k所有可以走到的点。h[k]存储这个单链表的头结点
2 int h[N], e[N], ne[N], idx;
3
4 // 添加一条边a->b
5 void add(int a, int b)
6 {
7     e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
8 }
9
10 // 初始化
11 idx = 0;
12 memset(h, -1, sizeof h);
```

## 树与图的遍历

时间复杂度 $O(n + m)$ ,  $n$ 表示点数,  $m$ 表示边数

(1) 深度优先遍历 —— 模板题 AcWing 846. 树的重心

```
1 int dfs(int u)
2 {
3     st[u] = true; // st[u] 表示点u已经被遍历过
4
5     for (int i = h[u]; i != -1; i = ne[i])
6     {
7         int j = e[i];
8         if (!st[j]) dfs(j);
9     }
10 }
```

(2) 宽度优先遍历 —— 模板题 AcWing 847. 图中点的层次

```
1 queue<int> q;
2 st[1] = true; // 表示1号点已经被遍历过
3 q.push(1);
4
5 while (q.size())
6 {
7     int t = q.front();
```

```

8     q.pop();
9
10    for (int i = h[t]; i != -1; i = ne[i])
11    {
12        int j = e[i];
13        if (!st[j])
14        {
15            st[j] = true; // 表示点j已经被遍历过
16            q.push(j);
17        }
18    }
19 }

```

## 拓扑排序 —— 模板题 AcWing 848. 有向图的拓扑序列

时间复杂度 $O(n + m)$ ,  $n$ 表示点数,  $m$ 表示边数

```

1  bool topsort()
2  {
3      int hh = 0, tt = -1;
4
5      // d[i] 存储点i的入度
6      for (int i = 1; i <= n; i ++ )
7          if (!d[i])
8              q[ ++ tt] = i;
9
10     while (hh <= tt)
11     {
12         int t = q[hh ++ ];
13
14         for (int i = h[t]; i != -1; i = ne[i])
15         {
16             int j = e[i];
17             if (-- d[j] == 0)
18                 q[ ++ tt] = j;
19         }
20     }
21
22     // 如果所有点都入队了, 说明存在拓扑序列; 否则不存在拓扑序列。
23     return tt == n - 1;
24
25 }

```

## 朴素dijkstra算法 —— 模板题 AcWing 849. Dijkstra求最短路 I

时间复杂是 $O(n^2 + m)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1 int g[N][N]; // 存储每条边
2 int dist[N]; // 存储1号点到每个点的最短距离
3 bool st[N]; // 存储每个点的最短路是否已经确定
4
5 // 求1号点到n号点的最短路, 如果不存在则返回-1
6 int dijkstra()
7 {
8     memset(dist, 0x3f, sizeof dist);
9     dist[1] = 0;
10
11     for (int i = 0; i < n - 1; i ++ )
12     {
13         int t = -1; // 在还未确定最短路的点中, 寻找距离最小的点
14         for (int j = 1; j <= n; j ++ )
15             if (!st[j] && (t == -1 || dist[t] > dist[j]))
16                 t = j;
17
18         // 用t更新其他点的距离
19         for (int j = 1; j <= n; j ++ )
20             dist[j] = min(dist[j], dist[t] + g[t][j]);
21
22         st[t] = true;
23     }
24
25     if (dist[n] == 0x3f3f3f3f) return -1;
26     return dist[n];
27 }
```

## 堆优化版dijkstra —— 模板题 AcWing 850. Dijkstra求最短路 II

时间复杂度 $O(m \log n)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1 typedef pair<int, int> PII;
2
3 int n; // 点的数量
4 int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
5 int dist[N]; // 存储所有点到1号点的距离
6 bool st[N]; // 存储每个点的最短距离是否已确定
7
8 // 求1号点到n号点的最短距离, 如果不存在, 则返回-1
9 int dijkstra()
10 {
11     memset(dist, 0x3f, sizeof dist);
12     dist[1] = 0;
13     priority_queue<PII, vector<PII>, greater<PII>> heap;
14     heap.push({0, 1}); // first存储距离, second存储节点编号
15
16     while (heap.size())
17     {
```

```

18     auto t = heap.top();
19     heap.pop();
20
21     int ver = t.second, distance = t.first;
22
23     if (st[ver]) continue;
24     st[ver] = true;
25
26     for (int i = h[ver]; i != -1; i = ne[i])
27     {
28         int j = e[i];
29         if (dist[j] > distance + w[i])
30         {
31             dist[j] = distance + w[i];
32             heap.push({dist[j], j});
33         }
34     }
35 }
36
37 if (dist[n] == 0x3f3f3f3f) return -1;
38 return dist[n];
39
40 }

```

## Bellman-Ford算法 —— 模板题 AcWing 853. 有边数限制的最短路

时间复杂度 $O(nm)$ ,  $n$ 表示点数,  $m$ 表示边数

注意在模板题中需要对下面的模板稍作修改, 加上备份数组, 详情见模板题。

```

1  int n, m;          // n表示点数, m表示边数
2  int dist[N];      // dist[x]存储1到x的最短路距离
3
4  struct Edge       // 边, a表示出点, b表示入点, w表示边的权重
5  {
6      int a, b, w;
7  }edges[M];
8
9  // 求1到n的最短路距离, 如果无法从1走到n, 则返回-1。
10 int bellman_ford()
11 {
12     memset(dist, 0x3f, sizeof dist);
13     dist[1] = 0;
14
15     // 如果第n次迭代仍然会松弛三角不等式, 就说明存在一条长度是n+1的最短路径, 由抽屉原理, 路径
16     // 中至少存在两个相同的点, 说明图中存在负权回路。
17     for (int i = 0; i < n; i ++ )
18     {
19         for (int j = 0; j < m; j ++ )
20         {
21             int a = edges[j].a, b = edges[j].b, w = edges[j].w;

```

```

21         if (dist[b] > dist[a] + w)
22             dist[b] = dist[a] + w;
23     }
24 }
25
26 if (dist[n] > 0x3f3f3f3f / 2) return -1;
27 return dist[n];
28 }

```

## spfa 算法 (队列优化的Bellman-Ford算法) —— 模板题 AcWing 851. spfa求最短路

时间复杂度平均情况下 $O(m)$ , 最坏情况下 $O(nm)$ ,  $n$ 表示点数,  $m$ 表示边数

```

1  int n;        // 总点数
2  int h[N], w[N], e[N], ne[N], idx;    // 邻接表存储所有边
3  int dist[N];    // 存储每个点到1号点的最短距离
4  bool st[N];    // 存储每个点是否在队列中
5
6  // 求1号点到n号点的最短路距离, 如果从1号点无法走到n号点则返回-1
7  int spfa()
8  {
9      memset(dist, 0x3f, sizeof dist);
10     dist[1] = 0;
11
12     queue<int> q;
13     q.push(1);
14     st[1] = true;
15
16     while (q.size())
17     {
18         auto t = q.front();
19         q.pop();
20
21         st[t] = false;
22
23         for (int i = h[t]; i != -1; i = ne[i])
24         {
25             int j = e[i];
26             if (dist[j] > dist[t] + w[i])
27             {
28                 dist[j] = dist[t] + w[i];
29                 if (!st[j])    // 如果队列中已存在j, 则不需要将j重复插入
30                 {
31                     q.push(j);
32                     st[j] = true;
33                 }
34             }
35         }
36     }
37
38     if (dist[n] == 0x3f3f3f3f) return -1;

```

```
39     return dist[n];
40 }
```

## spfa判断图中是否存在负环 —— 模板题 AcWing 852. spfa判断负环

时间复杂度是 $O(nm)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1  int n;        // 总点数
2  int h[N], w[N], e[N], ne[N], idx;    // 邻接表存储所有边
3  int dist[N], cnt[N];    // dist[x]存储1号点到x的最短距离, cnt[x]存储1到x的最短路中
   经过的点数
4  bool st[N];    // 存储每个点是否在队列中
5
6  // 如果存在负环, 则返回true, 否则返回false。
7  bool spfa()
8  {
9      // 不需要初始化dist数组
10     // 原理: 如果某条最短路径上有n个点(除了自己), 那么加上自己之后一共有n+1个点, 由抽屉原理
   一定有两个点相同, 所以存在环。
11
12     queue<int> q;
13     for (int i = 1; i <= n; i ++ )
14     {
15         q.push(i);
16         st[i] = true;
17     }
18
19     while (q.size())
20     {
21         auto t = q.front();
22         q.pop();
23
24         st[t] = false;
25
26         for (int i = h[t]; i != -1; i = ne[i])
27         {
28             int j = e[i];
29             if (dist[j] > dist[t] + w[i])
30             {
31                 dist[j] = dist[t] + w[i];
32                 cnt[j] = cnt[t] + 1;
33                 if (cnt[j] >= n) return true;    // 如果从1号点到x的最短路中包含
   至少n个点(不包括自己), 则说明存在环
34                 if (!st[j])
35                 {
36                     q.push(j);
37                     st[j] = true;
38                 }
39             }
40         }
41     }
```

```
42
43     return false;
44 }
```

## floyd算法 —— 模板题 AcWing 854. Floyd求最短路

时间复杂度是 $O(n^3)$ ,  $n$ 表示点数

```
1  初始化:
2      for (int i = 1; i <= n; i ++ )
3          for (int j = 1; j <= n; j ++ )
4              if (i == j) d[i][j] = 0;
5              else d[i][j] = INF;
6
7  // 算法结束后, d[a][b]表示a到b的最短距离
8  void floyd()
9  {
10     for (int k = 1; k <= n; k ++ )
11         for (int i = 1; i <= n; i ++ )
12             for (int j = 1; j <= n; j ++ )
13                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
14 }
```

## 朴素版prim算法 —— 模板题 AcWing 858. Prim算法求最小生成树

时间复杂度是 $O(n^2 + m)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1  int n;          // n表示点数
2  int g[N][N];    // 邻接矩阵, 存储所有边
3  int dist[N];    // 存储其他点到当前最小生成树的距离
4  bool st[N];     // 存储每个点是否已经在生成树中
5
6
7  // 如果图不连通, 则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
8  int prim()
9  {
10     memset(dist, 0x3f, sizeof dist);
11
12     int res = 0;
13     for (int i = 0; i < n; i ++ )
14     {
15         int t = -1;
16         for (int j = 1; j <= n; j ++ )
17             if (!st[j] && (t == -1 || dist[t] > dist[j]))
18                 t = j;
19
20         if (i && dist[t] == INF) return INF;
21
22         if (i) res += dist[t];
23         st[t] = true;
```

```

24
25     for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
26 }
27
28 return res;
29 }

```

## Kruskal算法 —— 模板题 AcWing 859. Kruskal算法求最小生成树

时间复杂度是 $O(m\log m)$ ,  $n$ 表示点数,  $m$ 表示边数

```

1  int n, m;          // n是点数, m是边数
2  int p[N];         // 并查集的父节点数组
3
4  struct Edge      // 存储边
5  {
6      int a, b, w;
7
8      bool operator< (const Edge &w) const
9      {
10         return w < W.w;
11     }
12
13 }edges[M];
14
15 int find(int x)    // 并查集核心操作
16 {
17     if (p[x] != x) p[x] = find(p[x]);
18     return p[x];
19 }
20
21 int kruskal()
22 {
23     sort(edges, edges + m);
24
25     for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集
26
27     int res = 0, cnt = 0;
28     for (int i = 0; i < m; i ++ )
29     {
30         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
31
32         a = find(a), b = find(b);
33         if (a != b)    // 如果两个连通块不连通, 则将这两个连通块合并
34         {
35             p[a] = b;
36             res += w;
37             cnt ++ ;
38         }
39     }
40

```



```
41     if (cnt < n - 1) return INF;
42     return res;
43 }
```

## 染色法判别二分图 —— 模板题 AcWing 860. 染色法判定二分图

时间复杂度是 $O(n + m)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1  int n;          // n表示点数
2  int h[N], e[M], ne[M], idx;    // 邻接表存储图
3  int color[N];   // 表示每个点的颜色, -1表示未染色, 0表示白色, 1表示黑色
4
5  // 参数: u表示当前节点, c表示当前点的颜色
6  bool dfs(int u, int c)
7  {
8      color[u] = c;
9      for (int i = h[u]; i != -1; i = ne[i])
10     {
11         int j = e[i];
12         if (color[j] == -1)
13             {
14                 if (!dfs(j, !c)) return false;
15             }
16         else if (color[j] == c) return false;
17     }
18
19     return true;
20 }
21
22
23 bool check()
24 {
25     memset(color, -1, sizeof color);
26     bool flag = true;
27     for (int i = 1; i <= n; i ++ )
28         if (color[i] == -1)
29             if (!dfs(i, 0))
30                 {
31                     flag = false;
32                     break;
33                 }
34     return flag;
35 }
```

## 匈牙利算法 —— 模板题 AcWing 861. 二分图的最大匹配

时间复杂度是 $O(nm)$ ,  $n$ 表示点数,  $m$ 表示边数

```
1 int n1, n2;    // n1表示第一个集合中的点数, n2表示第二个集合中的点数
2 int h[N], e[M], ne[M], idx;    // 邻接表存储所有边, 匈牙利算法中只会用到从第一个集合指向
   // 第二个集合的边, 所以这里只用存一个方向的边
3 int match[N];    // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
4 bool st[N];    // 表示第二个集合中的每个点是否已经被遍历过
5
6 bool find(int x)
7 {
8     for (int i = h[x]; i != -1; i = ne[i])
9     {
10         int j = e[i];
11         if (!st[j])
12         {
13             st[j] = true;
14             if (match[j] == 0 || find(match[j]))
15             {
16                 match[j] = x;
17                 return true;
18             }
19         }
20     }
21
22     return false;
23 }
24
25 // 求最大匹配数, 依次枚举第一个集合中的每个点能否匹配第二个集合中的点
26 int res = 0;
27 for (int i = 1; i <= n1; i ++ )
28 {
29     memset(st, false, sizeof st);
30     if (find(i)) res ++ ;
31 }
```

作者: yxc

链接: <https://www.acwing.com/blog/content/405/>

来源: AcWing

著作权归作者所有。商业转载请联系作者获得授权, 非商业转载请注明出处。