图的基本操作实现

图的操作所需要花费的时间通常既和顶点的个数n有关，又和边的条数m有关，因此时间复杂度会包含n和m两个变量。

- **邻接矩阵表示的图**

假设已知条件为：

图中实际顶点个数 *n_verts*、图中实际边的条数*m_edges*、

图中顶点可能的最大数量*kMaxVertex*、保存顶点数据的一维数组*ver_list*、

保存邻接矩阵内容的二维数组*edge_matrix*、

无边时权重的赋值*no_edge_value* (一般图为0，网为无穷大kMaxNum）、

有向或无向图标志*directed*(有向图为真，无向图为假)。

图用邻接矩阵表示时部分基本操作算法描述

算法7-1: 获取图的顶点个数 NumberOfVex(*graph*)

输入：图*graph*

输出：图的顶点个数

1.**return** *graph.n_verts*

时间复杂度 O（1）

算法7-2: 判断边是否存在 ExistEdge(graph, u, v)

输入：图graph、两个顶点u和v

输出：u到v有边返回 true，否则返回 false

1. **if** u < graph.*n_verts* 且 v < graph.*n_verts* **then**
2. | **if** u ≠ v 且 graph.edge_matrix[u][v]≠graph.*no_edge*_value **then**
3. | | **return true**
4. | end
5. end
6. **return false**

时间复杂度 O（1）

高等教育出版社

图用邻接矩阵表示时部分基本操作算法描述

算法7-4: 向图中插入边 InsertEdge(*graph, u,v,weight*)

输入：图*graph*，边的两个端点*u*和*v*，边的权重*weight*

输出：插入了边(*u,v*)或<*u,v*>的图

```
1. if u ≠ v 且 ExistEdge(graph, u, v) = false then
2. | graph.edge_matrix[u][v] ← weight
3. | graph.m_edges ← m_edges+1   //边数加1
4. | if graph.directed = false then  //如果是无向图，对主对角线对称的元素赋值
5. | | graph.edge_matrix[v][u] ← weight
6. | end
7. end
```

时间复杂度 O（1）

图用邻接矩阵表示时部分基本操作算法描述

算法7-6: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph*,*v*)

输入：图*graph*、顶点*v*

输出：删除了顶点*v*及所有邻接于顶点*v*的边的图*graph*

1. if $v < 0$ 或 $v \geq$ graph.n_verts then
2. | 待删除的顶点不存在，退出
3. end
4. *graph.ver_list*[*v*] ← *graph.ver_list*[*graph.n_verts-1*] //用最后一个顶点信息覆盖*v*

1-5： 时间复杂度 O（1）

高等教育出版社

图用邻接矩阵表示时部分基本操作算法描述

```
5.  count ← 0              //count计数由顶点v射出的边的条数
6.  for u = 0 to graph.n_verts-1 do   //遍历所有顶点
7.  |   if ExistEdge(graph, v, u) = true then  //存在边<v, u>
8.  |   | count ← count+1
9.  |   end
10. end
11. if graph.directed = true then   //有向图还要计数射入顶点v的边的条数?
12. |   for u = 0 to graph.n_verts-1 do
13. |   |   if ExistEdge(graph, u, v) = true then
14. |   |   | count ← count +1
15. |   |   end
16. |   end
17. end
```

5-10：时间复杂度 O（n）

11-17：时间复杂度 O（n）

图用邻接矩阵表示时部分基本操作算法描述

```
18. for u = 0 to graph.n_verts-1 do //将矩阵最后一行移入第v行
19. |      graph.edge_matrix[v][u] ← graph.edge_matrix[graph.n_verts-1][u]
20. end
21. for u = 0 to graph.n_verts-1 do //将矩阵最后一列移入第v列
22. |      graph.edge_matrix[u][v] ← graph.edge_matrix[u][graph.n_verts-1]
23. end
24. graph.m_edges ← graph.m_edges - count      //更新边的条数
25. graph.n_verts ← graph.n_verts - 1          //更新顶点个数
```

18-20： 时间复杂度 O（n）

21-23： 时间复杂度 O（n）

24-25： 时间复杂度 O（1）

加法原理，总时间复杂度 O（n）

高等教育出版社

图用邻接表表示时部分基本操作算法描述

算法7-7: 返回图中顶点的第一个邻接顶点 FirstAdjVex(*graph*,*v*)

输入： 图*graph*、顶点*v*

输出： 图*graph*中顶点*v*的第一个邻接顶点，若*v*无邻接顶点返回NIL。

1. **if** *v* < *graph.n_verts* **then**
2. | **return** *graph.ver_list*[*v*].*adj*
3. **end**

时间复杂度 O（1）

图用邻接表表示时部分基本操作算法描述

算法7-8: 判断边是否存在 ExistEdge(*graph, u, v*)
输入：图graph、两个顶点u和v
输出：u到v有边返回 true，否则返回 false

1. p ← FirstAdjVex(graph,u)　　//p指向u的第一个邻接结点
2. **while** p ≠ NIL 且 p.dest ≠ v **do**
3. | p ← p.next
4. **end**
5. **if** p≠NIL **then**
6. | **return true**
7. **else**
8. | **return false**
9. **end**

时间复杂度 O（m）

图用邻接表表示时部分基本操作算法描述

算法7-9: 向图中插入边 InsertEdge(*graph, u, v, weight*)

输入： 图*graph*，边的两个端点*u*和*v*，边的权重*weight*

输出： 插入了边(*u*,*v*)或<*u*,*v*>的图

```
1. if ExistEdge(graph, u, v) = false then
2. |  p ← new EdgeNode          //新结点链表
3. |  p.dest ← v                //从u指向v
4. |  p.weight ← weight
5. |  p.next ←graph.ver_list[u].adj
6. |  graph.ver_list[u].adj ← p  //新结点成为u的第一个邻接结点
7. |  graph.m_edges ←graph.m_edges + 1   //边数加1
```

时间复杂度 O（m）

时间复杂度 O（1）

思考：如果新结点每次都添加在单链表的末尾，能否实现O(1)的时间?

图用邻接表表示时部分基本操作算法描述

```
|  if graph.directed = false then    //如果是无向图，还要将u插入v的边表中
|  |  p ← new EdgeNode
|  |  p.dest ← u
|  |  p.weight ← weight
|  |  p.next ← graph.ver_list[v].adj
|  |  graph.ver_list[v].adj ← p
|  end
end
```

时间复杂度 O（1）

总的时间复杂度 O（m）

图用邻接表表示时部分基本操作算法描述

算法7-10: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph*,*v*)

输入： 图*graph*、顶点*v*

输出： 删除了顶点*v*及所有邻接于顶点*v*的边的图*graph*

```
1.  if v < 0 或 v ≥ graph.n_verts-1 then
2.  |  待删除的顶点不存在，退出
3.  end
4.  count ← 0    //count计数与顶点v邻接的边的条数
5.  p ← graph.ver_list[v].adj    //删除由顶点v射出的边
6.  while p ≠ NIL do
7.  | next_p ← p.next
8.  | delete p
9.  | count ← count + 1
10. | p ← next_p
11. end
```

1-11：时间复杂度 O（m）

图用邻接表表示时部分基本操作算法描述

算法7-10: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph,v*)

```
for u ← 0 to graph.n_verts-1 do //删除射入顶点v的边
|     p ← graph.ver_list[u].adj
|     if p ≠ NIL then    //非空链表
|  |     if p.dest = v then  //首结点为射入顶点v的边
|  |  |     graph.ver_list[u].adj ← p.next
|  |  |     delete p
|  |  |     count ← count+1
|  |     else //非首结点
|  |  |  while p.next ≠ NIL  且  p.next.dest ≠ v do //找到射入顶点v的边
|  |  |  |  p ← p.next
|  |  |  end
```

图用邻接表表示时部分基本操作算法描述

```
23. |   |   |   if p.next ≠ NIL then //找到<u,v>这条边，删除
24. |   |   |   |   next_p ← p.next
25. |   |   |   |   p.next ← next_p.next
26. |   |   |   |   delete next_p
27. |   |   |   |   count ← count+1
28. |   |   |   end
29. |   |   end
30. |   end
31. end
32. last_v ← graph.n_verts - 1 //最后一个顶点的编号
```

12-32：内外循环相关，换个角度分析，算法针对每个顶点，检测了其邻接的每一条边。故时间复杂度 O（n+m）

图用邻接表表示时部分基本操作算法描述

```
33. for u ← 0 to last_v-1 do   //将原来射入最后一个顶点的边都更新编号为v
34. |  p ← graph.ver_list[u].adj
35. |  if p ≠ NIL then    //非空链表
36. |  |  while p ≠ NIL  且  p.dest ≠ last_v do //找到射入顶点v的边
37. |  |  |  p ← p.next
38. |  |  end
39. |  |  if p ≠ NIL then //将原来射入最后一个顶点的边都更新编号为v
40. |  |  |  p.dest ← v
41. |  |  end
42. |  end
43. end
```

保持结点的编号连续！

33-43：和12-32同理，时间复杂度 O（n+m）

图用邻接表表示时部分基本操作算法描述

```
44. graph.ver_list[v] ← graph.ver_list[last_v] //顶点表中最后一个顶点移到位置v
45. if graph.directed = false then //无向图实际删除的边数要减半
46. |   count ← count / 2
47. end
48. graph.m_edges ← graph.m_edges – count //更新边数
49. graph.n_verts ← graph.n_verts-1  //更新顶点个数
```

44-49：时间复杂度 O（1）

加法原理：总时间复杂度 O（n+m）

图的遍历

按照某种方式逐个访问图中的所有顶点，且每个顶点只被访问一次。

1. 最简单的方式是沿着顶点表循环访问一遍，由此达到了遍历的目标。
   这种方式，完全没有借用边的信息。
2. 两种借助边信息（沿边）实现遍历的算法：**深度优先遍历**和**广度优先遍历**。

基于这两种遍历可以解决图中更多的涉及到边的问题，如图的连通性问题。

遍历图和遍历二叉树的不同

- 图中的顶点地位相同，没有特殊的顶点；二叉树结构中有一个特殊的根结点。
- 图中一个顶点可以和多个其它顶点邻接，可看作有多个直接前驱结点和多个后继结点，并可能存在回路。二叉树中每个结点的直接前驱结点只有一个，直接后继结点最多有两个，且不存在回路。
- 无向图中，邻接于一条边的两个顶点，甚至可以视作互为后继。

为避免通过不同前驱多次到达同一顶点，造成重复访问已经访问过的顶点，在图的遍历过程中，通常对已经访问过的顶点加特殊标记（即已访问标志）。

深度优先遍历

DFS（Depth First Search)

访问方式如下：

(1) 从选中的某一个未访问过的顶点出发，访问并对该顶点加已访问标志。

(2) 依次从该顶点的未被访问过的第1个、第2个、第3个…… 邻接顶点出发，依次进行深度优先遍历，即转向(1)。

(3) 如果还有顶点未被访问过，选中其中一个顶点作为起始顶点，再次转向(1)。如果所有的顶点都被访问到，遍历结束。

思考：什么条件下会从(3)再转向(1)?

深度优先遍历算法

算法7-11: 按深度优先遍历图中结点 DFS(*graph*)

输入： 图*graph*

输出： 图*graph*的深度优先遍历序列

关键数据结构： 顺序表visited

1. **for** $v \leftarrow 0$ **to** *graph.n_verts*-1 **do** //初始化各顶点的已访问标志为未访问
2. | *visited*[*v*] ← **false**
3. **end**
4. **for** $v \leftarrow 0$ **to** *graph.n_verts*-1 **do**
5. | **if** *visited*[*v*] = **false then**
6. | | DFS(*graph, v, visited*)
7. | **end**
8. **end**

1-3：时间复杂度 O（n）

4-8：for循环O(n), 循环体依赖于第6行DFS时间复杂度

深度优先遍历

算法7-12: 从指定顶点开始深度优先遍历 DFS(*graph,v,visited*)
输入： 图*graph*（邻接表存储），出发顶点*v*，已访问标志数组*visited*
输出： 图*graph*中从顶点*v*出发的深度优先访问序列

```
1. visited[v] ← true   //标记结点
2. Visit(graph, v)     //访问结点（抽象函数）
3. p ← graph.ver_list[v].adj
4. while p ≠ NIL do
5. |  if visited[p.dest] = false then   //邻接结点未访问
6. |  |  DFS(graph, p.dest, visited)   //递归访问邻接结点
7. |  end
8. |  p ← p.next   //继续遍历下一个邻接结点
9. end
```

1-9：每次DFS调用访问1个顶点和若干条边。合计访问到每个顶点和每条边一次，时间复杂度 O（n+m）。

深度优先遍历

算法7-12': 从指定顶点开始深度优先遍历 DFS(*graph,v,visited*)

输入：图*graph*（邻接矩阵存储），出发顶点*v*，已访问标志数组*visited*

输出：图*graph*中从顶点*v*出发的深度优先访问序列

```
visited[v] ← true   //标记结点
Visit(graph, v)     //访问结点（抽象函数）
for u ← 0 to graph.n_verts – 1 do
| if graph.edge_matrix[v][u] ≠ no_edge_value 且 visited[u] = false then
                                        //u是v的邻接结点，且未访问
| | DFS(graph, u, visited)  //递归访问邻接结点
| end
end
```

时间复杂度 O（n^2）

深度优先遍历算法的扩展

算法7-11: 按深度优先遍历图中结点 DFS(*graph*)

输入： 图*graph*

输出： 1. 各结点*u*第一次被“发现”的时刻 ***dfn***[*u*]

2. 各结点*u*遍历结束的时刻 ***fin***[*u*]

可用于割点、强连通成分等

3. 各结点*u*在深度优先遍历次序中的前驱结点 ***prev***[*u*] ---路径、生成树等

关键数据结构： 顺序表visited

全局变量： time 时间戳

深度优先遍历算法的扩展

算法： 按深度优先遍历图中结点 DFS(*graph*)

输入： 图*graph*

输出： 各结点首次发现时间，遍历结束时间，前驱结点

关键数据结构： 顺序表visited

全局变量： time 时间戳

```
1. for v ← 0 to graph.n_verts-1 do //初始化各顶点的已访问标志为未访问
2. |   visited[v] ← false |
3. end
4. time ← 1   //时间戳的初始值
5. for v ← 0 to graph.n_verts-1 do
6. |   if visited[v] = false then
7. |   |   prev[v] ← -1      //起点无前驱！
8. |   |   DFS(graph, v, visited)
9. |   end
10. end
```

高等教育出版社

深度优先遍历的扩展

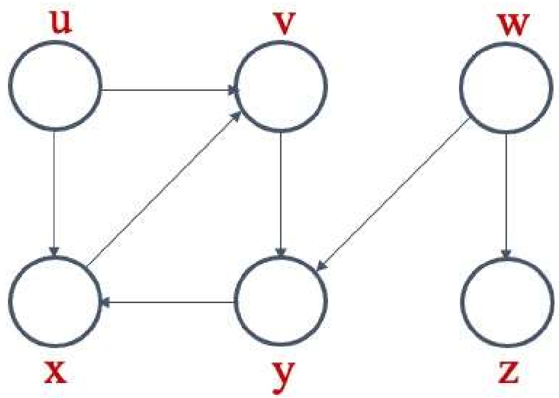
算法: 从指定顶点开始深度优先遍历 DFS(*graph,v,visited*)

```
1.  visited[v] ← true                 //标记结点
2.  dfn[v] ← time                     //记录第一次被发现的时间
3.  time ← time + 1                   //更新时间戳
4.  p ← graph.ver_list[v].adj
5.  while p ≠ NIL do
6.  |  if visited[p.dest] = false then   //邻接结点未访问
7.  |  |  prev[p.dest] ← v            //v成为其邻接结点的前驱
8.  |  |  DFS(graph, p.dest, visited) //递归访问邻接结点
9.  |  end
10. |  p ←p.next                      //继续遍历下一个邻接结点
11. end
12. fin[v] ← time                     //记录遍历结束的时间
13. time ← time + 1                   //更新时间戳
```

1-13：每次DFS调用访问1个顶点和若干条边。合计访问到每个顶点和每条边一次，时间复杂度 O（n+m）。

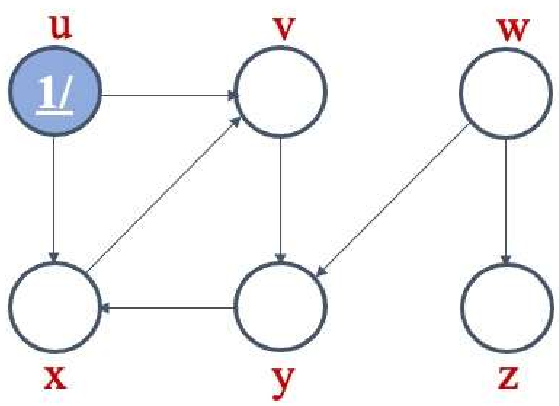
高等教育出版社

深度优先遍历的示例



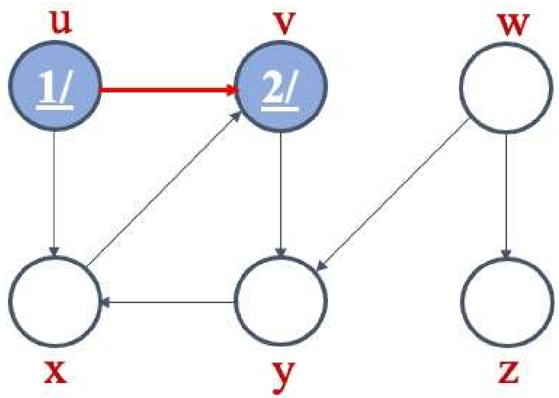
visited	F	F	F	F	F	F
prev						
	u	v	x	y	w	z

深度优先遍历的示例



visited	T	F	F	F	F	F
prev	-1					
	u	v	x	y	w	z

深度优先遍历的示例



	u	v	x	y	w	z
visited	T	T	F	F	F	F
prev	-1	u				

多选题 1分

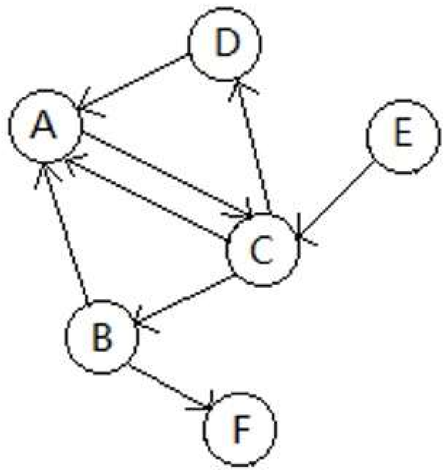
深度优先遍历右图，下面哪些遍历序列是合理的?

A <E, C, A, D, B, F>

B <A, C, B, F, D, E>

C <E, C, D, B, A, F>

D <B, A, C, D, F, E>



深度优先遍历的应用：查找路径

问题描述：给定图（有向或无向）和两个顶点 s, t, 判断图中是否存在从s到t的路径（连通性），如果s和t连通，则输出任意一条简单路径。

深度优先遍历的应用：查找路径

问题描述：给定图（有向或无向）和两个顶点 s, t, 判断图中是否存在从s到t的路径（连通性），如果s和t连通，则输出任意一条简单路径。

算法：FindPath(*graph*, *s*, *t*)
输入：图graph, 起点s，终点t
输出：如果存在从s到t的路径，则输出路径；否则输出 no path

```
for v ← 0 to graph.n_verts-1 do
|  visited[v] ← false
end
prev[s] ← -1   //起点无前驱
                              //从起点s出发深度优先查询
if DFS-Find(graph, s, t, visited) = true then  //s和t有路径连通
|  v ← t              //从终点开始，按倒序依次输出路径
|  while v ≥ 0 do         //prev[s] = -1
|  |  print v
|  |  v ← prev[v]     //走到前驱，继续输出
|  end
else          //s和t未连通
|  print "no path"
end
```

深度优先遍历的应用：查找路径

问题描述：给定图（有向或无向）和两个顶点 s, t, 判断图中是否存在从s到t的路径（连通性），如果s和t连通，则输出任意一条简单路径。

算法: DFS-Find(*graph, v, target, visited*)

```
visited[v] ← true   //标记结点
if v = target then
|  return true        //结点v就是目标结点，返回结果
end
p ← graph.ver_list[v].adj
while p ≠ NIL do
|  if visited[p.dest] = false then   //邻接结点未访问
|  |  prev[p.dest] ← v       //v成为其邻接结点的前驱
|  |  if DFS-Find(graph, p.dest, target, visited) = true then
|  |  |  return true       //从邻接结点出发找到目标结点
|  |  end                  //结束遍历
|  end
|  p ←p.next   //否则，继续遍历下一个邻接结点
end
return false   //从结点v出发未能找到到达目标结点的路径
```

深度优先遍历的应用：查找路径

问题描述：给定图（有向或无向）和两个顶点 s, t, 判断图中是否存在从s到t的路径（连通性），如果s和t连通，则输出任意一条简单路径。

- 时间复杂度：O(n+m)

算法：FindPath(graph, s, t)
输入：图graph, 起点s，终点t
输出：如果存在从s到t的路径，则输出路径；否则输出 no path

```
for v ← 0  to graph.n_verts-1 do
|   visited[v] ← false
end
prev[s] ← -1
                                    //从起点s出发深度优先查询
if DFS-Find(graph, s, t, visited) = true then
|   v ← t               //从终点开始，按倒序依次输出路径
|   while v ≥ 0 do          //起点s没有前驱，即prev[s] = -1
|   |   print v
|   |   v ← prev[v]      //走到前驱，继续输出
|   end
else            //s和t未连通
|   print "no path"
end
```

深度优先遍历的应用：查找路径

问题描述：给定图（有向或无向）和两个顶点 s, t, 判断图中是否存在从s到t的路径（连通性），如果s和t连通，则输出任意一条简单路径。

- 时间复杂度：O(n+m)

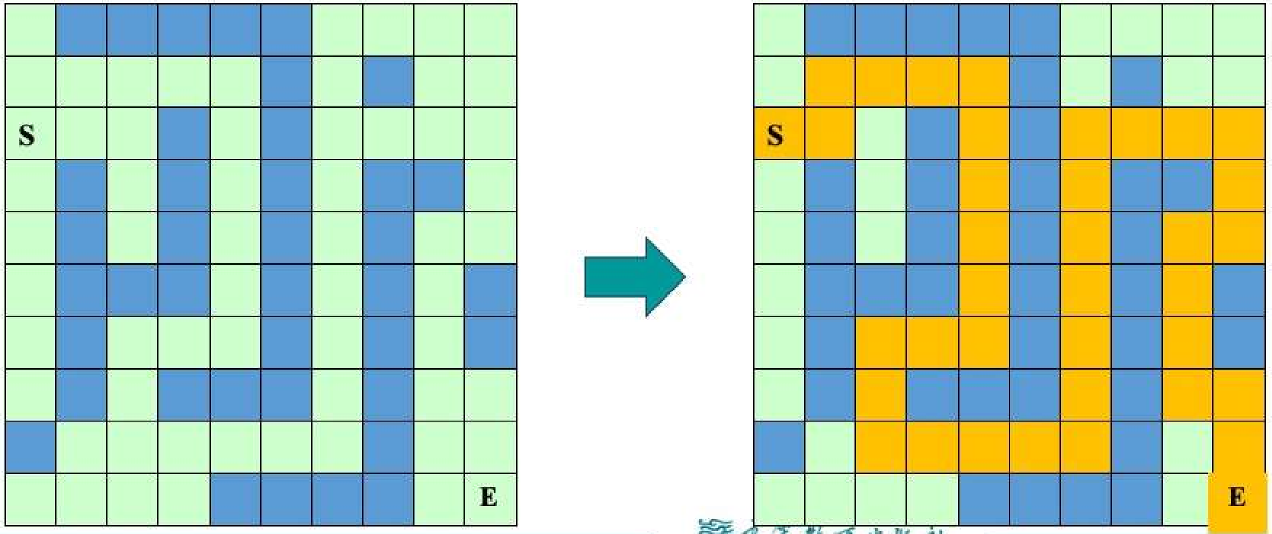
思考：如何查找从s到t的所有路径或者最短路径?

算法: DFS-Find(*graph, v, target, visited*)

```
visited[v] ← true   //标记结点
if v = target then
|  return true        //结点v就是目标结点，返回结果
end
p ← graph.ver_list[v].adj
while p ≠ NIL do
| if visited[p.dest] = false then  //邻接结点未访问
| | prev[p.dest] ← v       //v成为其邻接结点的前驱
| | if DFS-Find(graph, p.dest, target, visited) = true then
| | | return true      //从邻接结点出发找到目标结点
| | end                //结束遍历
| end
| p ←p.next   //否则，继续遍历下一个邻接结点
end
return false   //从结点v出发未能找到到达目标结点的路径
```

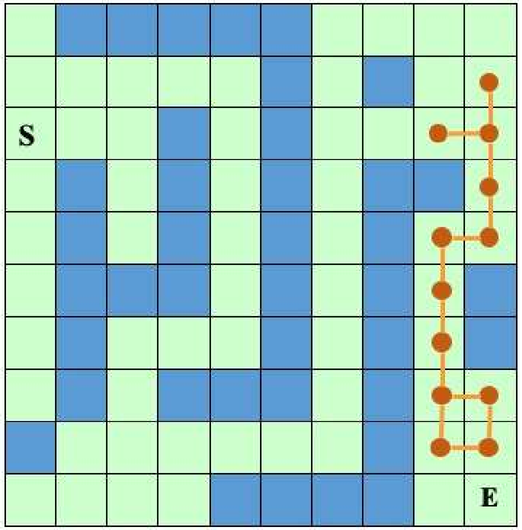
深度优先遍历的应用：走迷宫

问题描述：表示迷宫的二维数组M[1..n, 1..m], 其中M[i][j]=0表示方格(i, j)可以通过，而M[i][j]=1表示该方格设有障碍，不能通过($1 \leq i \leq n, 1 \leq j \leq m$)。从标有S的方格（起点）出发，每步可以移到当前位置上下左右相邻的、且可通过的方格，查找走到终点（标有E的方格）的路径。



深度优先遍历的应用：走迷宫

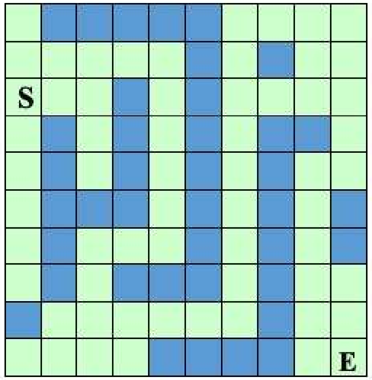
问题描述：表示迷宫的二维数组M[1..n, 1..m], 其中M[i][j]=0表示方格(i, j)可以通过，而M[i][j]=1表示该方格设有障碍，不能通过。从标有S的方格（起点）从发，每步可以移到当前位置上下左右相邻的、且可通过的方格，查找走到终点（标有E的方格）的路径。



- 可将迷宫（二维数组）转换为无向图
- 每个可通过的方格成为结点，与上下左右相邻且可通过的方格有边相连
- 走迷宫问题变成在无向图中查找从起点至终点的路径查找问题

思考：如何存储对应迷宫的无向图？需要用到邻接矩阵或邻接表吗？

深度优先遍历的应用：走迷宫



算法: DFS-FindMaze($M, n, m, x, y, t_x, t_y, visited$)

输入： 二维数组M[1..n, 1..m], 当前位置(x, y), 终点(t_x, t_y)

输出： 查找从起点至终点的路径

关键数据结构： adj ← {(0, -1), (0, 1), (-1, 0), (1, 0)} //上下左右的（相对）位置

```
visited[x][y] ← true   //标记当前单元格位置，这里visited和prev都是二维表格
if x = t_x 且 y = t_y then
|  return true        //走到目标，返回结果
end
for k ← 0 to 3 do       //依次查找上下左右相邻位置
|  nx ← x + adj[k][0]
|  ny ← y + adj[k][1]                //相邻位置有效、非障碍且未被访问过
|  if 1 ≤ nx ≤ n 且 1 ≤ ny ≤ m 且 M[nx][ny] ≠ 1 且 visited[nx][ny] = false then
|  |  prev[nx][ny] ← k        //只需记录与前驱结点(x,y)的相对位置！？
|  |  if DFS-FindMaze(M, nx, ny, t_x, t_y, visited) = true then
|  |  |  return true
|  |  end
|  end
end
return false
```

- 时间复杂度：O(mn)
- 空间复杂度：O(mn)

广度优先遍历

BFS（Breadth First Search）

访问方式如下：

(1) 从选中的某一个未访问过的顶点出发，访问并对该顶点加已访问标志

(2) 依次对该顶点的未被访问过的第1个、第2个、第3个……第 k 个邻接点 v_1 、v_2、v_3…… v_k进行访问且加已访问标志

(3) 依次对顶点 v_1 、v_2、v_3…… v_k转向操作(2)

(4) 如果还有顶点未被访问过，选中其中一个顶点作为起始顶点，再次转向(1)。如果所有的顶点都被访问到，遍历结束

广度优先遍历

BFS（Breadth First Search)

每个结点要经历的三个阶段（状态）

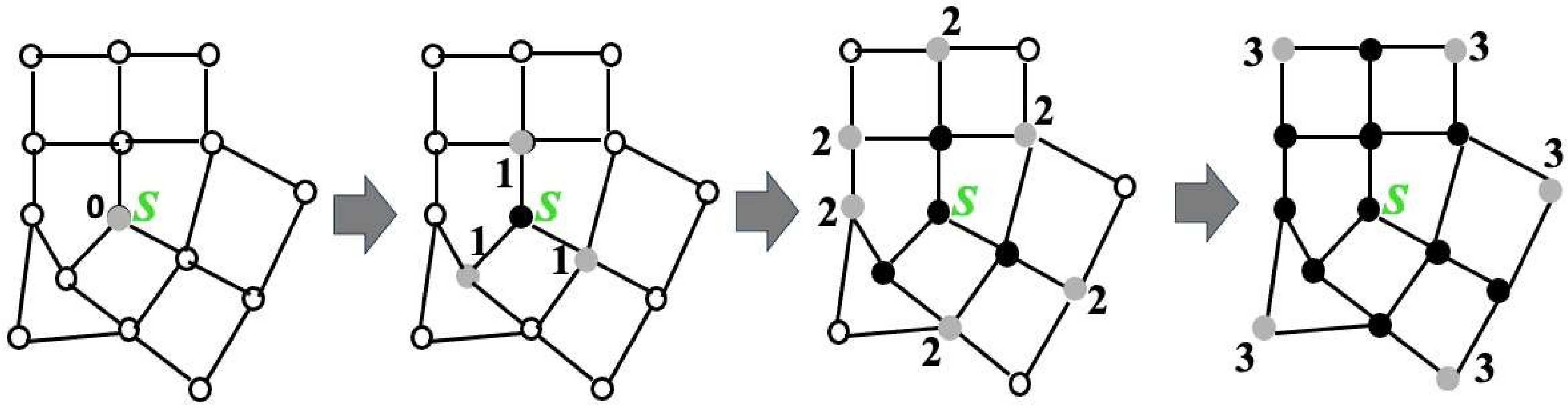
- 未被发现 (Undiscovered)
- 被发现（Discovered)
- 遍历结束（Finished)

结点及其所有邻接结点已被发现

广度优先遍历

BFS（Breadth First Search）

- Undiscovered
- Discovered
- Finished



数值表示从起点到各结点的最短路径长度！ （边无权重或权重均为1）

广度优先遍历算法

<u>**算法7-13:** 按广度优先遍历图中结点 BFS(*graph*)</u>

输入： 图*graph*

输出： 图*graph*的广度优先遍历序列

```
1.  for v ← 0 to graph.n_verts-1 do //初始化各顶点的已访问标志为未访问
2.  |  visited[v] ← false
3.  end
4.  for v ← 0  to graph.n_verts-1 do
5.  |  if visited[v] = false then
6.  |  |  visited[v] ← true    //标记起点
7.  |  |  prev[v] ← -1   //起点无前驱
8.  |  |  dist[v] ← 0   //离起点的最小距离（最少边数）
9.  |  |  BFS(graph, v, visited)
10. |  end
11. end
```

1-3：时间复杂度 O（n）

4-11：依赖于第6行BFS时间复杂度

广度优先遍历算法

算法7-14: 按广度优先遍历图中结点 BFS(*graph, v, visited*)

输入： 图*graph*，出发顶点*v*，已访问标志数组*visited*

输出： 图*graph*中从顶点*v*出发的广度优先遍历序列

```
InitQueue(queue)
EnQueue(queue, v)    //起点入队
while IsEmpty(queue) = false do
|  u ← DeQueue(queue)
|  Visit(graph, u)             //访问结点
|  p ← graph.ver_list[u].adj
|  while p ≠ NIL do         //遍历邻接结点
|  |  if visited[p.dest] = false then   //邻接结点未发现！      Undiscovered
|  |  |  visited[p.dest] ← true
|  |  |  prev[p.dest] ← u
|  |  |  dist[p.dest] ← dist[u] + 1   //设置离起点的最小距离      Discovered
|  |  |  EnQueue(queue, p.dest)   //邻接结点入队
|  |  end
|  end          Finished
end
```

广度优先遍历算法

算法7-14: 按广度优先遍历图中结点 BFS(*graph, v, visited*)

输入： 图*graph*，出发顶点*v*，已访问标志数组*visited*

输出： 图*graph*中从顶点*v*出发的广度优先遍历序列

```
1.  InitQueue(queue)
2.  EnQueue(queue, v)   //起点入队
3.  while IsEmpty(queue) = false do
4.  |  u ← DeQueue(queue)
5.  |  Visit(graph, u)            //访问结点
6.  |  p ← graph.ver_list[u].adj
7.  |  while p ≠ NIL do           //遍历邻接结点
8.  |  |  if visited[p.dest] = false then  //邻接结点未发现！
9.  |  |  |  visited[p.dest] ← true
10. |  |  |  prev[p.dest] ← u
11. |  |  |  dist[p.dest] ← dist[u] + 1   //设置离起点的最小距离
12. |  |  |  EnQueue(queue, p.dest)   //邻接结点入队
13. |  |  end
14. |  end
15. end
```

- 每个结点入队一次、出队一次
- 每个结点被查询（第8行代码）的次数等于其入度（？）
- 时间复杂度：O(n+m)
- 空间复杂度：O(n)？