实验二

1. 实验题目

通信网络上的线路设计问题

1. 实验目的
2. 掌握无向连通图生成树的求解方法；
3. 掌握基本回路系统和环路空间的求解方法；
4. 掌握基本割集系统和断集空间的求解方法；
5. 了解生成树、环路空间和断集空间的实际应用。
6. 实验要求

在通信网络中，节点之间有网络线路传输数据包，假设两节点之间最多有一条网络线路，给定该通信网络节点之间的连接关系，求解如下问题：

1. 用图结构描述上述通信网络，如给定相邻矩阵：。
2. 如何在通信网络中确保消息可以有效地从源节点传播到所有目标节点，以减少了冗余的数据转发？（提示：通过构建一棵生成树，数据可以沿着生成树的拓扑结构进行传输（如给出生成树的关联矩阵）。）
3. 有多少种上述实现方案？（提示：求方阵的行列式和秩见参考代码。）
4. 如何在该通信网络中，解决数据包转发和路由选择的问题？（提示：通过探测和避免环路空间，可以保证数据在网络中的正常传输，并降低数据包丢失和延迟。即输出基本回路系统（输出形式如：{e1e4e3,e2e5e3}）和环路空间（输出形式如：{Φ,e1e4e3,e2e5e3,e1e4e5e2}）。）
5. 在网络安全和隐私保护中，如何识别可能容易遭到攻击或数据泄露的路径或节点？（通过分析断集空间，可以采取相应的保护措施，提高网络安全性和数据隐私性。即输出基本割集系统（输出形式如：{{e1,e4},{e2,e5},{e3,e4,e5}}）和断集空间（输出形式如：{Φ,{e1,e4},{e2,e5},{e3,e4,e5},{e1,e2,e4,e5},{e1,e3,e5},{e2,e3,e4},{e1,e2,e3}}）。）
6. 实验内容和实验步骤
7. 需求分析
8. 输入的形式和输入值的范围：

输入形式：例如

0 1 1 0 1

1 0 1 0 1

1 1 0 1 0

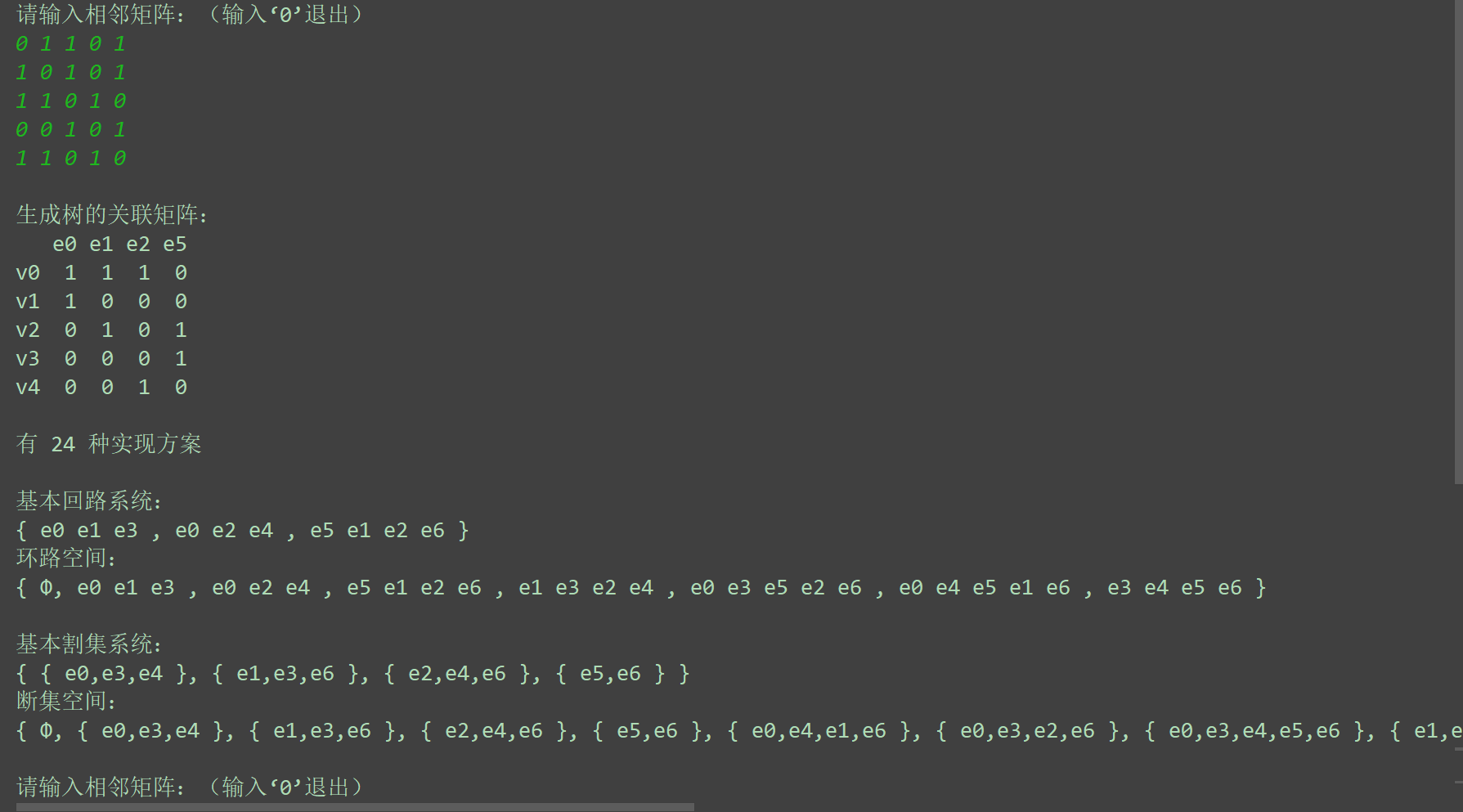
0 0 1 0 1

1 1 0 1 0

（给定的相邻矩阵）

输入值的范围：非负整数

1. 输出的形式：



如图所示，输出时，按照实验要求的顺序依次展开输出

─生成树的关联矩阵

─方案数

─基本回路系统和环路空间

─基本割集系统和断集空间

1. 程序所能实现的功能：

* 能通过给定的相邻矩阵输出生成树的关联矩阵，计算生成树的个数，输出基本回路系统和环路空间，输出基本割集系统和断集空间
* 可循环，反复输入，方便测试多组数据

1. 概要设计

我选择的是Python进行编程，

下面是各函数的功能分析：

1. 生成树的关联矩阵

先计算出关联矩阵，然后去除关联矩阵中的任意一行，取出一个结果，其中从m中选出的n-1列就是选出的树枝。采用二进制枚举进行排列组合，判断1~2n+1-1转换成二进制数有多少个1，符合则将它存起来。

1. 计算生成树的个数，即实现方案数

设图G的度矩阵为D(G)，邻接矩阵为A(G)。定义拉普拉斯矩阵L(G) = D(G) - A(G)，则L(G)的任意一个余子式的值即为图G的生成树个数。通过分析，我们发现拉普拉斯矩阵的对角线上的元素是相邻矩阵对应行上所有元素的度数之和，而其他元素为相邻矩阵的相反数。得到拉普拉斯矩阵后，我们可以取其第一行第一列的余子式，即去除第一行和第一列的行列式。

对于求行列式的过程，我们可以采用定义法，按照第一行展开，得到n个代数余子式。这些余子式同样是求行列式，然后按照它们的第一行展开，一直递归到只剩一行时结束，即可计算出整个行列式的值。

1. 求此生成树对应的基本回路系统

通过分析生成树的边，我们可以推导出与该生成树相关联的弦。接着，根据每条弦连接的两个顶点，在生成树中找到这两个顶点之间的一条路径。将这条路径与弦一起组合，形成一个回路。这样，我们就能得到基本回路系统。在寻找路径的过程中，采用了广度优先搜索（BFS）算法，以生成树中的某一顶点作为根节点，搜索时记录其子节点。最后，根据保存的子节点信息，为每个子节点找到其对应的父节点。通过以另一个顶点为起点，向上回溯直到根节点，我们就能得到一条完整的回路。

1. 求此生成树对应的基本割集系统

对于每一条树枝，可以先将其删除，然后尝试将一条弦加入图中。如果加入弦之后，图仍然是连通的，那么该弦应该属于与这条树枝关联的割集；否则，该弦不属于该割集。连通性的判断可以通过对图的邻接矩阵进行广度优先搜索（BFS）来实现，如果所有节点都被访问过，则图是连通的。

1. 环路空间和断集空间

环路空间的计算可以通过选择生成树的基本回路系统中的若干个回路（1~n）进行环合运算来实现。具体过程是先选取第一个回路生成数组A，然后对于后续的每个回路，检查其中的边是否在A中。如果在A中，则将这条边从A中删除，如果不在A中，则将这条边加入A。

为了实现取若干个回路的操作，可以使用二进制枚举法，从1到2n-1，其中1表示被选中，0表示不被选中。通过这种方式，可以枚举出所有选择的情况。在这里，可以使用 itertools 中的 combination 函数来实现。

割集空间的计算与环路空间类似，同样可以通过选择生成树的基本回路系统中的若干个割集进行割集合并运算来得到结果。

1. 详细设计

源代码：

1. **import** numpy as np
2. **import** copy
3. **import** itertools

6. # 关联矩阵
7. **def** Incidence\_matrix(l: list) -> list:
8. ll = copy.deepcopy(l)
9. m = []
10. **for** i **in** range(len(l)):
11. **for** j **in** range(len(l)):
12. **if** ll[i][j] > 0:
13. **for** k **in** range(ll[i][j]):
14. t = [0] \* len(l)
15. t[i], t[j] = 1, 1
16. m.append(t)
17. ll[i][j] -= 1
18. ll[j][i] -= 1
19. **return** m

22. # 生成树
23. **def** tree(l: list) -> tuple:
24. x = list(itertools.combinations(range(len(l)), len(l[0]) - 1))
25. side = []
26. **for** each **in** x:
27. matrix = [l[i][1:] **for** i **in** each]
28. **if** np.linalg.det(matrix) != 0:
29. side = each
30. **break**
31. adj\_tree = [[0] \* len(l[0]) **for** \_ **in** range(len(l[0]))]
32. inc\_tree = [l[i] **for** i **in** side]
33. **for** i **in** inc\_tree:
34. e = [j **for** j **in** range(len(i)) **if** i[j] == 1]
35. adj\_tree[e[0]][e[1]], adj\_tree[e[1]][e[0]] = 1, 1
36. **return** adj\_tree, inc\_tree, side

39. # 计算生成树的个数
40. **def** tree\_number(l: list) -> int:
41. m = []
42. **for** i **in** range(1, len(l)):
43. t = []
44. **for** j **in** range(1, len(l)):
45. **if** i == j:
46. t.append(sum(l[i]))
47. **else**:
48. t.append(-l[i][j])
49. m.append(t)
50. **return** int(round(np.linalg.det(np.array(m)), 0))

53. # 定义节点类
54. **class** Node:
55. **def** \_\_init\_\_(self, value, pre=None):
56. self.pre = pre
57. self.value = value
58. self.succ = []

61. # 广度优先搜索
62. **def** BFS\_loop(adj\_tree: list, start: int, end: int):
63. nodes = [Node(i) **for** i **in** range(len(adj\_tree))]
64. visited = [False] \* len(adj\_tree)
65. visited[start] = True
66. queue = [start]
67. # BFS搜索
68. **while** queue:
69. now = queue.pop(0)
70. **if** now == end:
71. **break**
72. **for** i **in** range(len(adj\_tree)):
73. **if** adj\_tree[now][i] == 1 **and** visited[i] == False:
74. nodes[now].succ.append(nodes[i])
75. queue.append(i)
76. visited[i] = True
77. # 为所有后继节点的pre赋值，方便找到父节点
78. **for** node **in** nodes:
79. **for** i **in** node.succ:
80. i.pre = node
81. path = [end]
82. # 从终点向回找，找到根节点结束
83. **while** nodes[end].pre:
84. end = nodes[end].pre.value
85. path.insert(0, end)
86. **return** path

89. # 基本回路系统
90. **def** Basic\_loop\_system(adj\_tree: list, inc: list, side: list):
91. chord = list(set(range(len(inc))) - set(side))
92. loop\_point = []
93. # 计算每条弦的回路中的点
94. **for** each **in** chord:
95. e = [i **for** i **in** range(len(inc[0])) **if** inc[each][i] == 1]
96. path = BFS\_loop(adj\_tree, e[0], e[1])
97. path.append(e[0])
98. loop\_point.append(path)
99. loop\_side = []
100. # 根据回路中的点求出对应的边
101. **for** each **in** loop\_point:
102. index = 1
103. x = []
104. **while** index < len(each):
105. t = [0] \* len(inc[0])
106. t[each[index - 1]], t[each[index]] = 1, 1
107. **for** i **in** range(len(inc)):
108. **if** inc[i] == t:
109. x.append('e' + str(i) + ' ')
110. **break**
111. index += 1
112. loop\_side.append(x)
113. **return** loop\_side

116. # 判断是否连通
117. **def** Connected(m: list) -> bool:
118. visited = [False] \* len(m)
119. visited[0] = True
120. queue = [0]
121. # BFS搜索
122. **while** queue:
123. now = queue.pop(0)
124. **for** i **in** range(len(m)):
125. **if** m[now][i] == 1 **and** visited[i] == False:
126. queue.append(i)
127. visited[i] = True
128. **return** all(visited)

131. # 基本割集系统
132. **def** Basic\_cutset\_system(adj\_tree: list, side: list, inc: list):
133. chord = list(set(range(len(inc))) - set(side))
134. cutset\_side = []
136. **for** i **in** range(len(side)):
137. adj\_tree\_copy = copy.deepcopy(adj\_tree)
138. # 删除树枝
139. e = [\_ **for** \_ **in** range(len(inc[0])) **if** inc[side[i]][\_] == 1]
140. adj\_tree\_copy[e[0]][e[1]], adj\_tree\_copy[e[1]][e[0]] = 0, 0
141. x = ['e' + str(side[i])]
142. **for** each **in** chord:
143. # 加上一条弦
144. e = [\_ **for** \_ **in** range(len(inc[0])) **if** inc[each][\_] == 1]
145. adj\_tree\_copy[e[0]][e[1]], adj\_tree\_copy[e[1]][e[0]] = 1, 1
146. **if** Connected(adj\_tree\_copy):
147. # 如果连通则去除该弦，并加入割集
148. x.append('e' + str(each))
149. e = [\_ **for** \_ **in** range(len(inc[0])) **if** inc[each][\_] == 1]
150. adj\_tree\_copy[e[0]][e[1]], adj\_tree\_copy[e[1]][e[0]] = 0, 0
151. cutset\_side.append(x)
152. **return** cutset\_side

155. # 环路空间和断集空间
156. **def** space(side: list):
157. rep = copy.deepcopy(side)
158. **for** i **in** range(2, len(side) + 1):
159. x = itertools.combinations(range(len(side)), i)
160. **for** e **in** x:
161. t = copy.deepcopy(side[e[0]])
162. **for** j **in** e[1:]:
163. **for** each **in** side[j]:
164. **if** each **in** t:
165. t.remove(each)
166. **else**:
167. t.append(each)
168. **if** t **not** **in** rep:
169. rep.append(t)
170. **return** rep



175. #主函数
176. **while** True:
177. **print**("请输入相邻矩阵：（输入‘0’退出）")
178. adj = []
179. cin = input()
181. **if** cin == "0":
182. **break**
184. **while** cin:
185. adj.append(list(map(int, cin.split())))
186. cin = input()
188. inc = Incidence\_matrix(adj)
189. adj\_tree, inc\_tree, side = tree(inc)
191. **print**("生成树的关联矩阵：")
192. **print**("   ", end="")
193. **for** i **in** side:
194. **print**('e' + str(i), end=" ")
195. **print**()
196. **for** i **in** range(len(inc\_tree[0])):
197. **print**("v" + str(i), end="  ")
198. **for** j **in** range(len(inc\_tree)):
199. **print**(inc\_tree[j][i], end="  ")
200. **print**()
202. tree\_num = tree\_number(adj)
203. **print**(f"\n有 {tree\_num} 种实现方案\n")
205. loop\_side = Basic\_loop\_system(adj\_tree, inc, side)
206. **print**("基本回路系统：")
207. **print**("{ ", end="")
208. **for** i **in** range(len(loop\_side)):
209. **if** i != len(loop\_side) - 1:
210. **print**(''.join(loop\_side[i]), end=", ")
211. **else**:
212. **print**(''.join(loop\_side[i]), end="")
213. **print**("}")
215. beltway = space(loop\_side)
216. **print**("环路空间：")
217. **print**("{ Φ, ", end="")
218. **for** i **in** range(len(beltway)):
219. **if** i != len(beltway) - 1:
220. **print**(''.join(beltway[i]), end=", ")
221. **else**:
222. **print**(''.join(beltway[i]), end="")
223. **print**("}")
225. **print**()
227. cutset\_side = Basic\_cutset\_system(adj\_tree, side, inc)
228. **print**("基本割集系统：")
229. **print**("{ ", end="")
230. **for** i **in** range(len(cutset\_side)):
231. **print**("{ ", end="")
232. **if** i != len(cutset\_side) - 1:
233. **print**(','.join(cutset\_side[i]), end=" }, ")
234. **else**:
235. **print**(','.join(cutset\_side[i]), end=" }")
236. **print**(" }")
238. breakset = space(cutset\_side)
239. **print**("断集空间：")
240. **print**("{ Φ, ", end="")
241. **for** i **in** range(len(breakset)):
242. **print**("{ ", end="")
243. **if** i != len(breakset) - 1:
244. **print**(','.join(breakset[i]), end=" }, ")
245. **else**:
246. **print**(','.join(breakset[i]), end=" }")
247. **print**(" }")
249. **print**()
250. 调试分析
251. 调试过程中所遇到的问题及解决方法：

* 部分环路空间输出不全

解决方法：暂未解决

* 未熟练掌握BFS

解决方法：通过网上的素材借鉴理解学习

1. 算法的时空分析：

1. Incidence\_matrix函数

时间复杂性：

* + **O(n^3)，其中 n 为相邻矩阵的边数。主要由三层循环嵌套引起，其中第一和第二层的长度均为 n，第三层的长度最坏为 n。**

空间复杂性：

* + **O(n^2)，由于需要存储生成的关联矩阵。**

2. tree函数

时间复杂性：

* + **O(n^4)，主要由两层嵌套循环引起。内部循环中包含对 numpy 的 det 操作，其时间复杂度为 O(n^3)。**

空间复杂性：

* + **O(n^2)，由于需要存储生成的关联矩阵。**

3. tree\_number函数

时间复杂性：

* + **O(n^3)，由于需要计算 numpy 的 det 操作。**

空间复杂性：

* + **O(n^2)，由于需要存储生成的关联矩阵。**

4. Node类

时间复杂性：

* + **构造函数为 O(1)。**

空间复杂性：

* + **O(1)。**

5. BFS\_loop函数

时间复杂性：

* + **O(n^2)，由于广度优先搜索中的两层嵌套循环。**

空间复杂性：

* + **O(n)，存储节点和队列。**

6. Basic\_loop\_system函数

时间复杂性：

* + **O(n^4)，包含两层嵌套循环。内部循环中包含对 BFS\_loop 函数的调用，其时间复杂度为 O(n^2)。**

空间复杂性：

* + **O(n^2)，存储回路点和回路边。**

7. Connected函数

时间复杂性：

* + **O(n^2)，由于广度优先搜索中的两层嵌套循环。**

空间复杂性：

* + **O(n)，存储访问标记和队列。**

8. Basic\_cutset\_system函数

时间复杂性：

* + **O(n^4)，包含两层嵌套循环。内部循环中包含对 Connected 函数的调用，其时间复杂度为 O(n^2)。**

空间复杂性：

* + **O(n^2)，存储割集边。**

9. space函数

时间复杂性：

* + **O(2^n \* m)，其中 n 为回路的数量，m 为回路平均长度。由于使用了组合生成函数，会生成指数级的组合情况。**

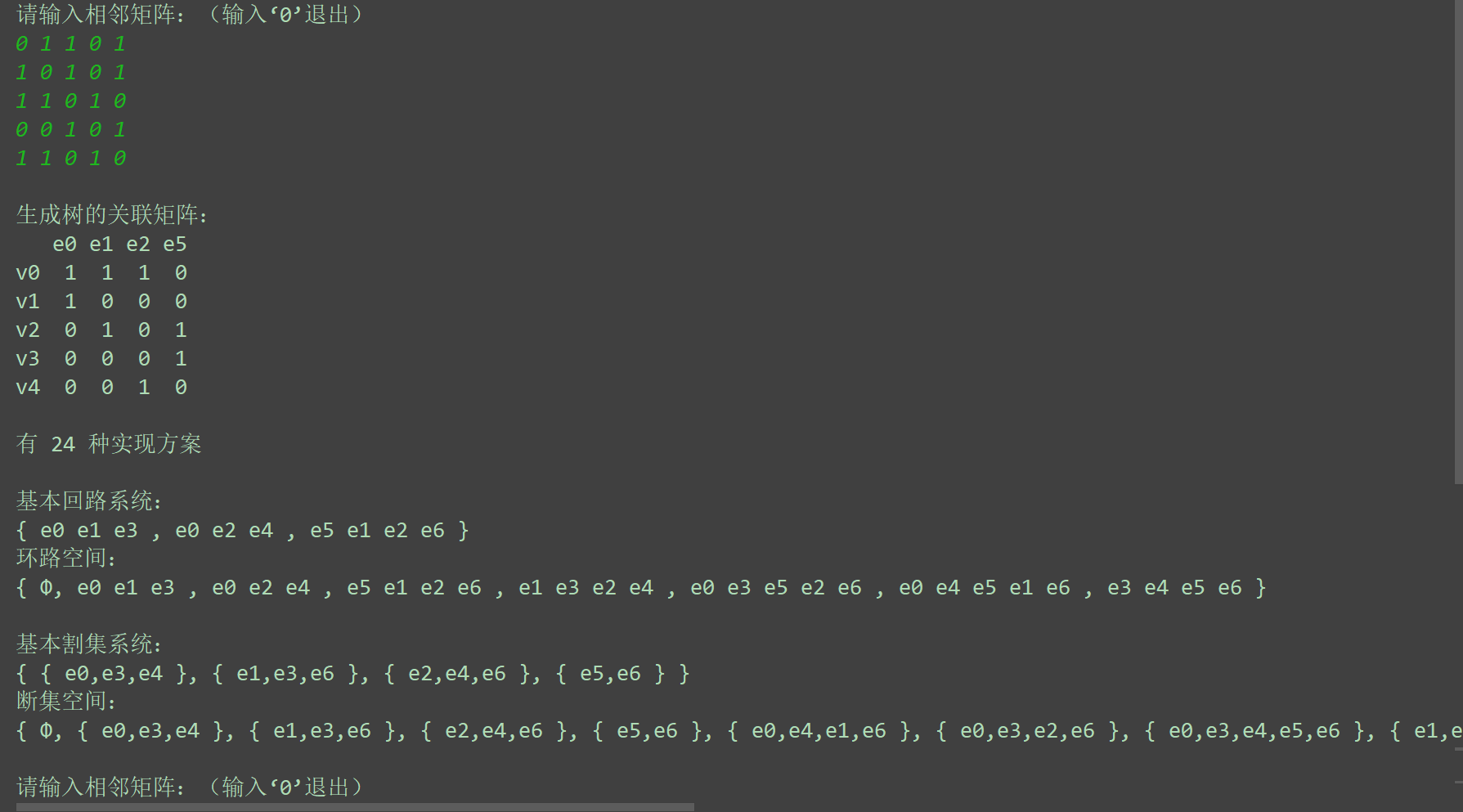
空间复杂性：

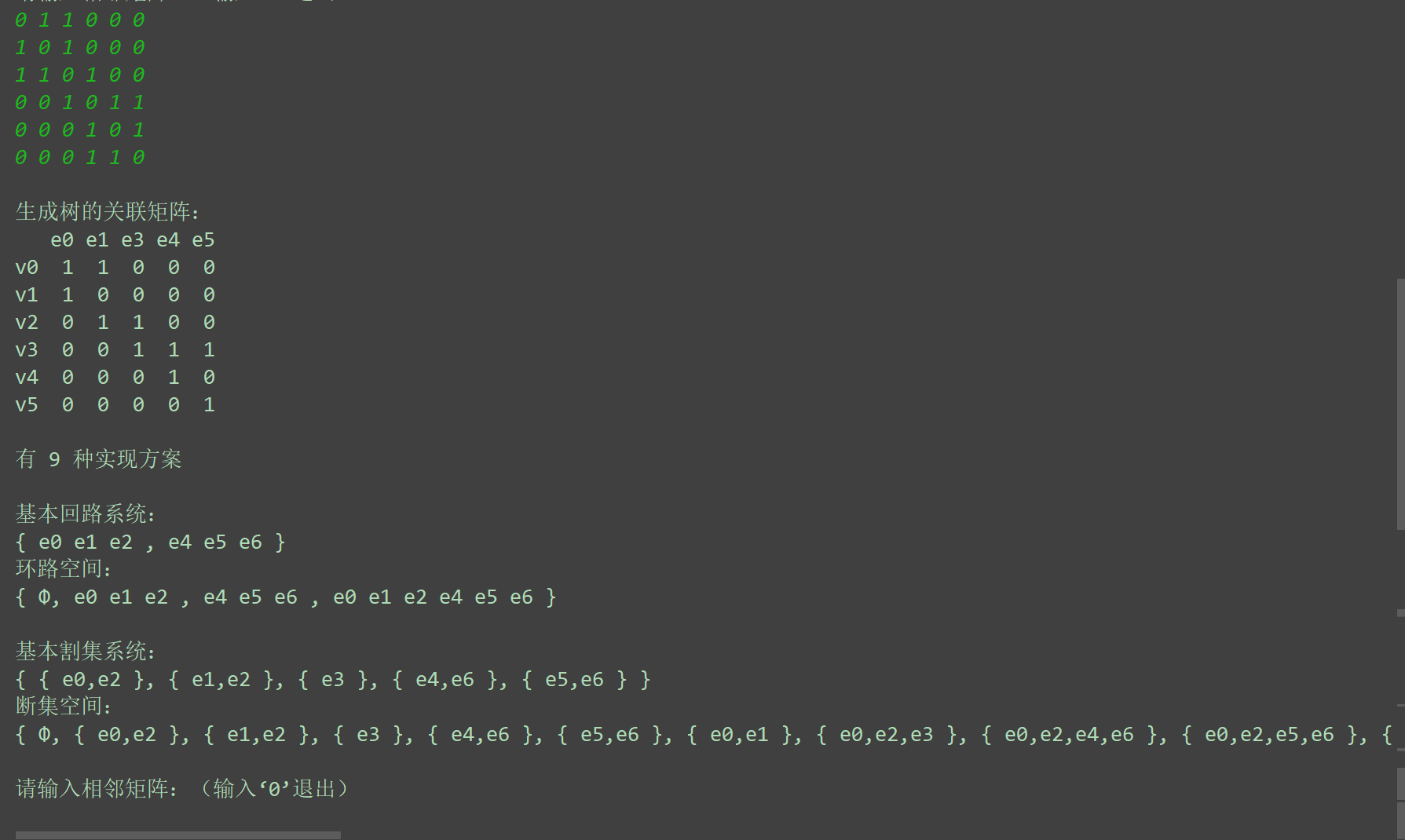
* + **O(2^n \* m)，存储生成的环路空间。**

10. 整体而言：

* + **这个程序的复杂度主要受到生成树和回路系统的计算部分的影响。由于广度优先搜索和组合生成函数的使用，一些函数的时间复杂度可能较高，特别是在处理大规模图的情况下。因此，在实际应用中，可能需要进一步优化算法或采用更高效的数据结构和算法。**
* 实际性能可能会受到不同输入情况的影响。

1. 实验结果





1. 实验总结

* 实验过程中的感悟和体会：

1. 理解图论概念：

在编写程序之前，我花了一些时间深入理解了图论的基本概念，包括关联矩阵、生成树、回路等。这为编写相应的算法提供了基础。

1. 算法设计思路：

实现图论算法需要清晰的思路和设计。我通过分析问题，将整个流程拆分成多个函数，每个函数负责一个具体的功能。这种模块化的设计使得代码更清晰、易读且易于维护。

1. 广度优先搜索：

在实现基本回路系统和基本割集系统时，我使用了广度优先搜索算法。这个算法对于遍历图的特定部分非常有效，对于实现问题的解决方案提供了重要的帮助。

1. 异常处理的重要性：

在实现生成树和相关计算时，我充分利用了 numpy 库提供的矩阵操作功能。这使得矩阵计算更为简便，加速了程序的执行。

* 经验和教训：

1. 函数模块化设计：

在设计程序时，将大问题拆分成小的、相对独立的函数是一个良好的实践。这样不仅提高了代码的可读性，而且降低了出错的可能性。

1. 异常处理：

在用户输入部分，我没有添加足够的异常处理机制。在实际应用中，用户可能输入不符合要求的数据，因此要保证程序能够健壮地处理各种输入情况。

1. 代码注释：

在一些复杂的算法部分，我添加了注释来解释代码的逻辑和功能。这对于其他人阅读和理解代码非常有帮助，也有助于自己在未来回顾代码时更容易理解。

1. 优化思考：

部分算法的时间复杂度可能较高，特别是当图较大时。在实际应用中，可能需要考虑进一步的优化策略，例如改进算法或者使用更高效的数据结构。

* 总结：

通过这次实验，我不仅学到了图论算法的实现方法，还提高了对算法设计和代码优化的认识。编写图论算法需要对问题有深入的理解，而模块化的设计和代码注释是确保代码质量的关键。

在未来的工作中，我将更注重对算法复杂度的优化，并且在程序中添加更多的异常处理机制，以提高程序的稳定性。此外，我还计划进一步学习和掌握图论领域的其他算法，以便更好地解决更复杂的问题。这次实验为我打开了图论算法领域的大门，为我的学术和职业发展提供了有益的经验。