**算法设计与分析第八章作业**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **姓名** | **王靳** | **班级** | **计科十班** | **学号** | 220111012 |
| **第1题** |  | | | | |
| **第2题** |  | | | | |
| **第3题** |  | | | | |
| **第4题** |  | | | | |
| **总分** |  | | | | |
| **备注** | 作业提交截止时间：2023 年11 月18 日24:00，超过提交截至时间的作业视为无效。确因网络等特殊原因无法及时提交作业的学生，应至少提前1小时与助教联系沟通。作业提交邮箱：\_hitsz\_algorithm@126.com\_。作业文件名命名方式： 第x章-x班-姓名-学号（例，第1章-1班-张三-220110101.docx）； 邮件主题为：第x章作业, x班，姓名，学号（例，第1章作业，1班，张三，220110101）。缺少这些信息的作业将被酌情扣分。 | | | | |

1. 最短路径问题

采用本章所学的最短路径算法，编程完成以下题目，并给出采用该种算法的原因。

**题目1：**

给定一个n个点m条边的有向图，图中可能存在重边和自环，边权可能为负数。请你求出从1号点到n号点的最多经过k条边的最短距离，如果无法从1号点走到n号点，输出impossible。

**输入格式：**

第一行包含三个整数n，m，k。接下来m行，每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

**输出格式：**

输出一个整数，表示从1号点到n号点的最多经过k条边的最短距离。如果不存在满足条件的路径，则输出impossible。

**输入样例：**

3 3 1

1 2 1

2 3 1

1 3 3

**输出样例：**

3

这里使用了bellman-ford的一种改进的版本shortest-path-fast-algorithmn(spfa)，平均情况下的时间复杂度是O(m + n), 最坏情况下的时间复杂度是O(mn)。这里之所以使用bellman-ford，因为题目说明了“边权可能是负数”。我这里也采用了spfa的一种优化，就是让较小的元素放在队列的前面。

1. from collections import deque  *# 双端队列, append, appendleft, pop, popleft*
2. import math
3. def spfa(graph: list[list], s: int, k: int) -> list:
4. *# s 是 源点*
5. *# 只对经过路径不超过k的时松弛*
6. num = len(graph)
7. path\_len = [0] \* num  *# 路径的长度不超过k*
8. d = [float("inf")] \* num  *# 初始化距离*
9. d[s] = 0
10. q = deque()  *# 初始化双端队列*
11. q.append(s)
12. while len(q) != 0:
13. u = q.popleft()
14. if path\_len[u] >= k:
15. continue
16. else:
17. for v in range(num):  *# 遍历邻居*
18. if path\_len[v] >= k:
19. continue
20. elif d[u] + graph[u][v] < d[v]:
21. d[v] = d[u] + graph[u][v]  *# 松弛*
22. path\_len[v] = path\_len[u] + 1
23. if not v in q:
24. q.append(v)
25. if d[q[-1]] < d[q[0]]:  *# 优化, 较小者优先*
26. temp = q.popleft()
27. q.append(temp)
28. *# 判断是否有负环, 最后一轮松弛*
29. for i in range(num):
30. if path\_len[i] >= k:
31. continue
32. else:
33. for j in range(num):
34. if path\_len[j] >= k:
35. continue
36. elif d[i] + graph[i][j] < d[j]:
37. return None
38. return d
39. def solution(n: int, m: int, k: int, edges: list):
40. *# m条边, n个节点, 路径最多是k*
41. *# edges中的元素是(start, end, weight)*
42. graph = [[float("inf")] \* n for \_ in range(n)]
43. for e in edges:
44. start, end, weight = e
45. graph[start - 1][end - 1] = weight
46. d = spfa(graph, 0, k)
47. if d == None:
48. print("imposible")
49. elif math.isinf(d[n - 1]):
50. print("imposible")
51. else:
52. print(d[n - 1])
53. if \_\_name\_\_ == "\_\_main\_\_":
54. n = 3
55. m = 3
56. k = 1
57. edges = [(1, 2, 1), (2, 3, 1), (1, 3, 3)]
58. solution(n, m, k, edges)

**题目2：**

A国有N个城市，编号为1、…、N。小明是编号为1的城市中一家公司的员工，今天突然接到了上级通知需要去编号为N的城市出差。

由于疫情原因，很多直达的交通方式暂时关闭，小明无法乘坐飞机直接从城市1到达城市N,需要通过其他城市进行陆路交通中转。小明通过交通信息网，查询到了M条城市之间仍然还开通的路线信息以及每一条路线需要花费的时间。

同样由于疫情原因，小明到达一个城市后需要隔离观察一段时间才能离开该城市前往其他城市。通过网络，小明也查询到了各个城市的隔离信息。（由于小明之前在城市1，因此可以直接离开城市1，不需要隔离）

由于上级要求，小明希望能够尽快赶到城市N，因此他求助于你，希望你能帮他规划一条路线，能够在最短时间内到达城市N。

**输入格式：**

第1行：两个正整数N，M，N表示A国的城市数量，M表示末关闭的路线数量

第2行：N个正整数，第i个整数Ci表示到达编号为i的城市后需要隔离的时间

第3至M+2行：每行3个正整数，u，v，c，表示有一条城市u到城市v的双向路线仍然开通着，通过该路线的时间为c。

**输出格式：**

第1行：1个正整数，表示小明从城市1出发到达城市N的最短时间。（到达城市N不需要计算城市N的隔离时间）

**输入样例：**

4 4

5 7 3 4

1 2 4

1 3 5

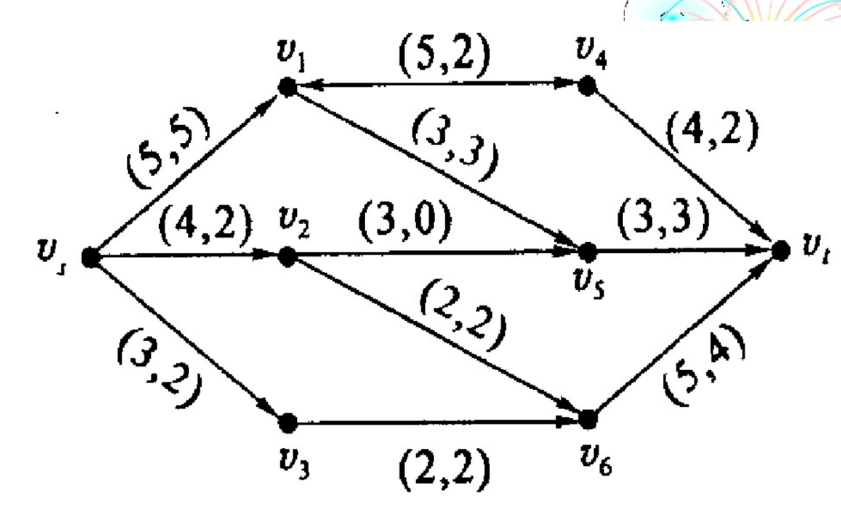
2 4 3

3 4 5

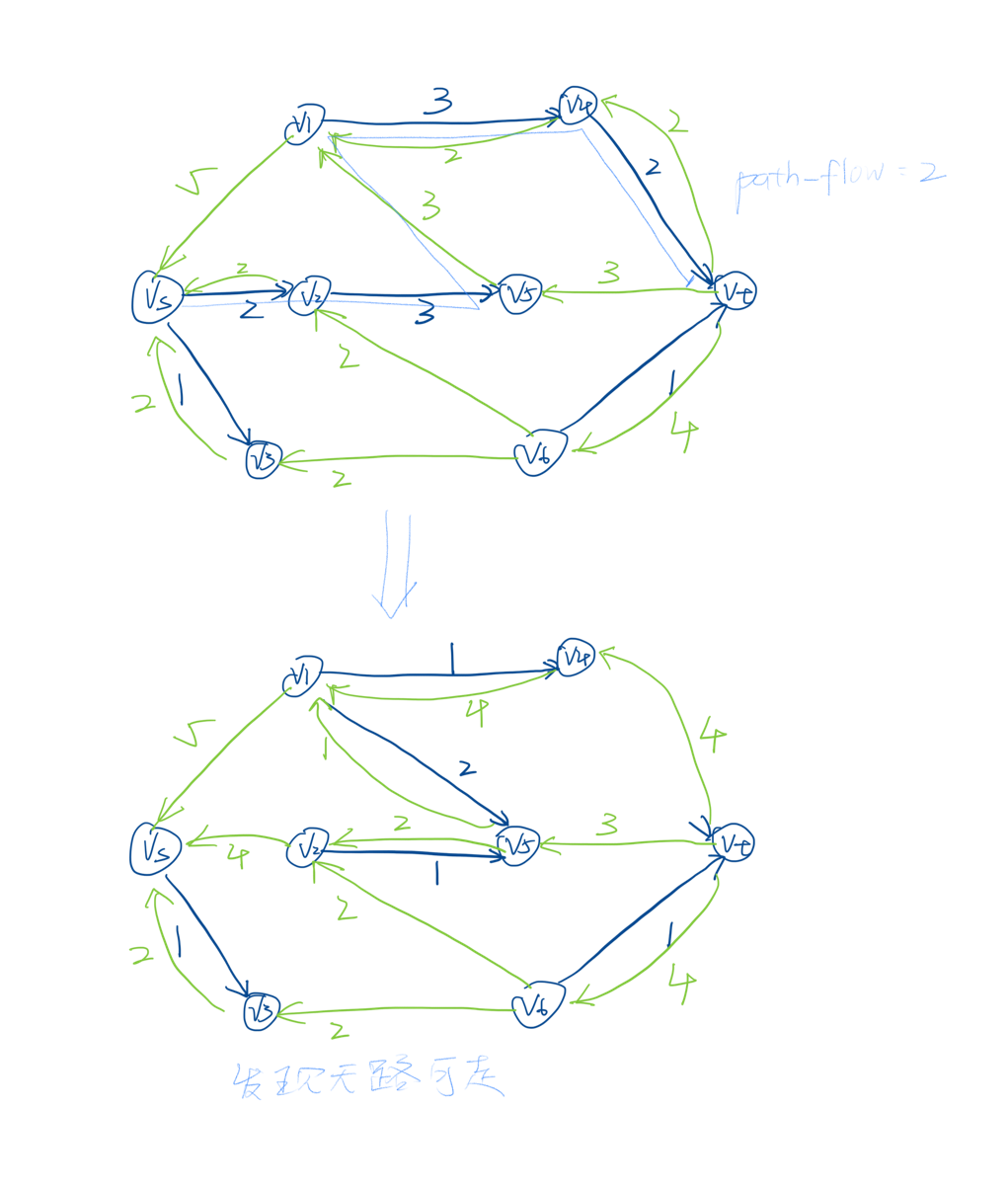
**输出样例：**

13

这里采用dijkstra算法，因为这里的边都是非负的。在松弛的时候，考虑在城市的滞留时间

1. import heapq  *# 可以使用元组, 这样就不会hash冲突了*
2. import math
3. def dijkstra(graph: list[list], stop: list, s: int, pre: list) -> list:
4. *# len(pre) = num*
5. num = len(graph)
6. d = [float("inf")] \* num
7. d[s] = 0
8. visited: set = set()
9. q: list = [(0, s)]  *# 元组会对第一个元素进行比较*
10. while len(q) != 0:
11. d\_u, u = heapq.heappop(q)  *# 得到最小*
12. visited.add(u)
13. for v in range(num):
14. if v in visited:
15. continue
16. elif d\_u + graph[u][v] + stop[u] < d[v]:  *# 松弛*
17. d[v] = d\_u + graph[u][v] + stop[u]
18. pre[v] = u
19. heapq.heappush(q, (d[v], v))
20. d = [i - stop[s] for i in d]  *# 不用在出发城市滞留*
21. return d
22. def solution(n: int, m: int, stop: list[int], edges: list):
23. graph = [[float("inf")] \* n for \_ in range(n)]
24. for e in edges:
25. start, end, weight = e
26. graph[start - 1][end - 1] = weight
27. pre = [-1] \* n
28. d: list = dijkstra(graph, stop, 0, pre)
29. return d[n - 1]
30. if \_\_name\_\_ == "\_\_main\_\_":
31. n = 4
32. m = 4
33. stop = [5, 7, 3, 4]
34. edges = [(1, 2, 4), (1, 3, 5), (2, 4, 3), (3, 4, 5)]
35. result = solution(n, m, stop, edges)
36. print(result)
37. 网络流问题  
    

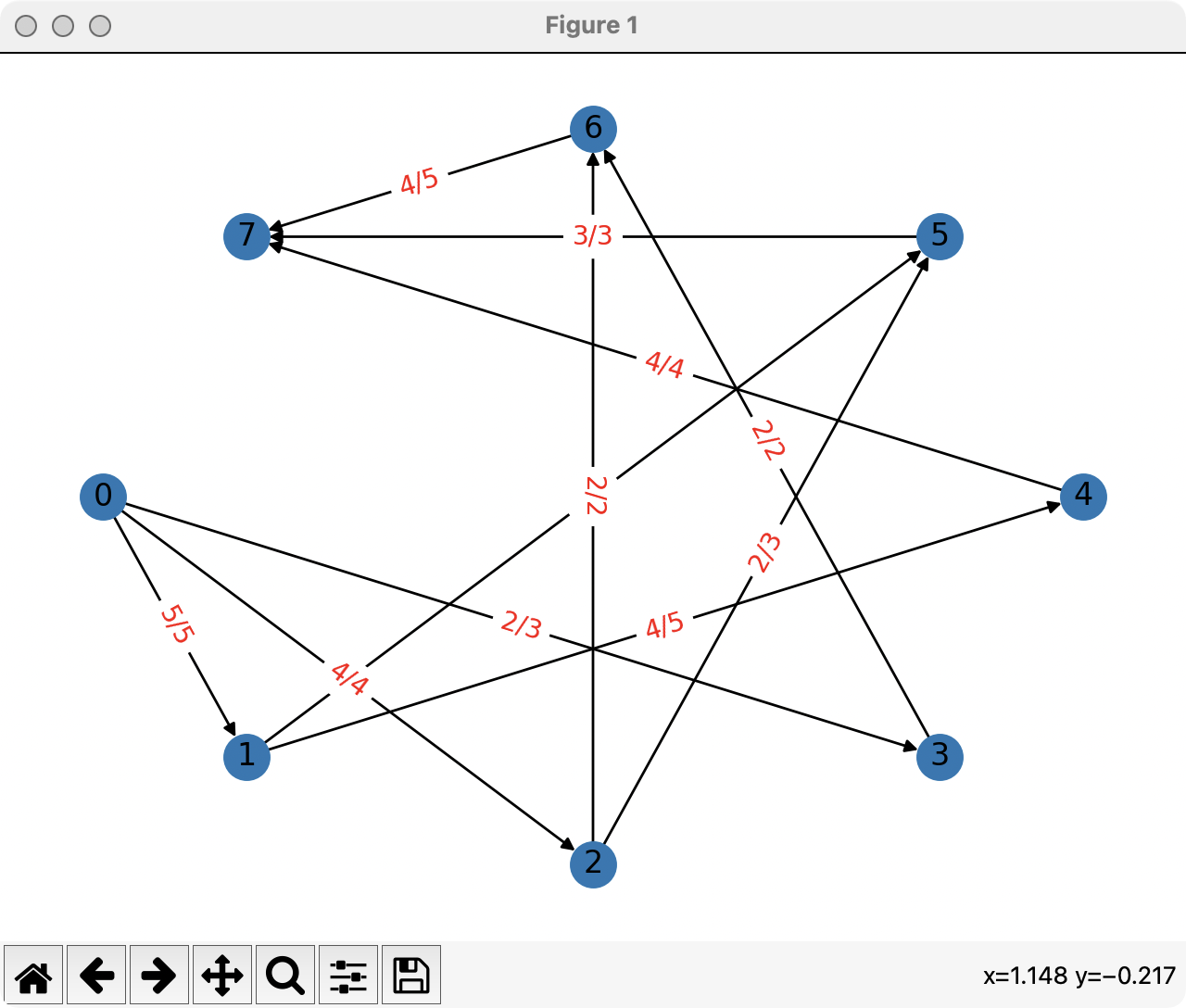
上图展示了一个网络，求这个网络的最大流。每条边上的前一个数字表示容量，后一个数字表示初始可行流。



得到最大流是5 + 4 + 2 = 11

1. import networkx as nx
2. import matplotlib.pyplot as plt
3. from collections import deque  *# 双端队列, popleft, append*
4. def draw\_result(adj\_matrix, flow):
5. num\_nodes = len(adj\_matrix)
6. G = nx.DiGraph()
7. for i in range(num\_nodes):
8. G.add\_node(i)
9. for i in range(num\_nodes):
10. for j in range(num\_nodes):
11. if adj\_matrix[i][j] != 0:
12. G.add\_edge(
13. i,
14. j,
15. weight="{}/{}".format(flow[i][j], adj\_matrix[i][j]),
16. )
17. *# 设置布局算法*
18. pos = nx.shell\_layout(G)
19. labels = {n: str(n) for n in G.nodes}
20. nx.draw(G, pos, with\_labels=True, label=labels)
21. edge\_labels = {(u, v): d["weight"] for u, v, d in G.edges(data=True)}
22. nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_color="red")
23. plt.title("result")
24. plt.show()
25. return G
26. class solution:
27. def \_\_init\_\_(self, n: int, edges: list[tuple], s: int, t: int) -> None:
28. self.graph = [[0] \* n for \_ in range(n)]
29. *# graph中, 反向的路径也就是正向的流量*
30. self.s = s
31. self.t = t
32. self.num = n
33. for e in edges:
34. start, end, weight = e
35. self.graph[start][end] = weight
36. def bfs(self, s: int, t: int, pre: list) -> bool:
37. *# 能从s到达t, 那么就返回true*
38. visited = [0] \* self.num
39. m\_q = deque()
40. m\_q.append(s)
41. visited[s] = True
42. while len(m\_q) != 0:
43. u = m\_q.popleft()
44. for v in range(self.num):
45. if not visited[v] and self.graph[u][v] > 0:
46. m\_q.append(v)
47. visited[v] = True
48. pre[v] = u
49. if v == t:
50. return True
51. return False
52. def FordFulkson(self):
53. pre = [-1] \* self.num
54. max\_flow = 0
55. while self.bfs(self.s, self.t, pre):
56. path\_flow = float("inf")  *# 瓶颈*
57. s = self.t
58. while s != self.s:
59. path\_flow = min(path\_flow, self.graph[pre[s]][s])
60. s = pre[s]
61. max\_flow += path\_flow
62. v = self.t
63. while v != self.s:
64. u = pre[v]
65. self.graph[u][v] -= path\_flow
66. self.graph[v][u] += path\_flow
67. v = pre[v]
68. return max\_flow
69. if \_\_name\_\_ == "\_\_main\_\_":
70. edges = [
71. (0, 1, 5),
72. (0, 2, 4),
73. (0, 3, 3),
74. (1, 4, 5),
75. *# (4, 1, 5),*
76. (1, 5, 3),
77. (2, 5, 3),
78. (2, 6, 2),
79. (3, 6, 2),
80. (6, 7, 5),
81. (5, 7, 3),
82. (4, 7, 4),
83. ]
84. n = 8
85. s = 0
86. t = 7
87. capa = [[0] \* n for \_ in range(n)]
88. for e in edges:
89. start, end, weight = e
90. capa[start][end] = weight
91. result = solution(n, edges, s, t)
92. print("max flow: ", result.FordFulkson())
93. flow = [[0] \* n for \_ in range(n)]
94. for i in range(n):
95. for j in range(n):
96. if capa[i][j] >= 0:
97. flow[i][j] = result.graph[j][i]
98. draw\_result(capa, flow)
99. *# print(capa)*
100. *# print(flow)*

输出结果：



max\_flow: 11

1. 匹配问题

**题目：**

若两个正整数的和为素数，则这两个正整数称之为“素数伴侣”，如2和5、6和13，它们能应用于通信加密。现在密码学会请你设计一个程序，从已有的 N （ N 为偶数）个正整数中挑选出若干对组成“素数伴侣”，挑选方案多种多样，例如有4个正整数：2，5，6，13，如果将5和6分为一组中只能得到一组“素数伴侣”，而将2和5、6和13编组将得到两组“素数伴侣”，能组成“素数伴侣”最多的方案称为“最佳方案”，当然密码学会希望你寻找出“最佳方案”。

**输入格式：**

第1行：一个正偶数n，表示待挑选的自然数个数。

第2行：n个正整数，表示待挑选的自然数。

**输出格式：**

输出一个整数 K ，表示你求得的“最佳方案”组成“素数伴侣”的对数。

**输入样例：**

4

2 5 6 13

**输出样例：**

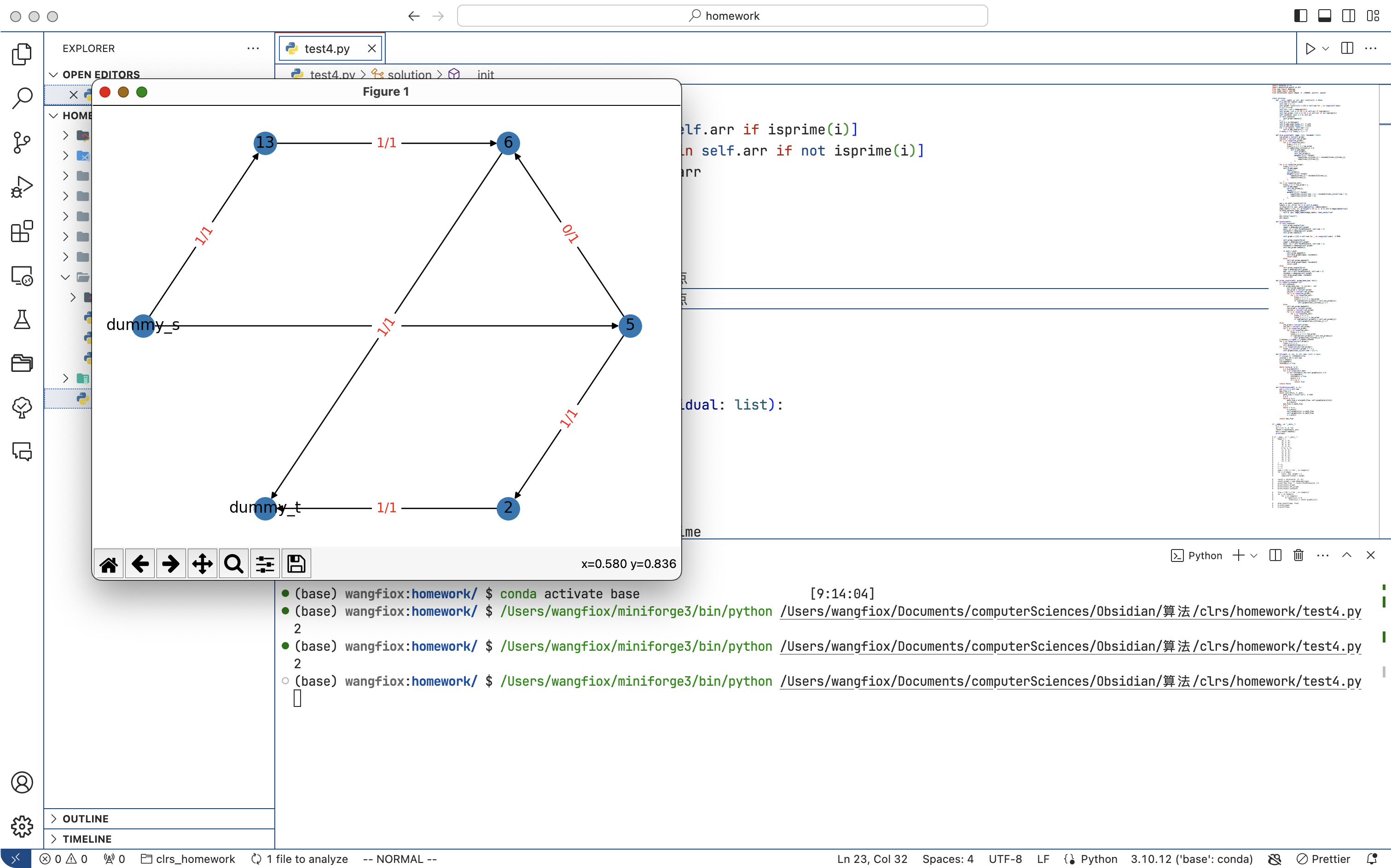
2

如果能组成“素数伴侣”（相加为素数），那么之间就可以有边。

素数除了2以外都是非偶数，因此素数（除了2），素数（除了2）之间是不能有边的，否则相加就会是偶数，偶数除了2不可能为素数，因此，可以划分为一个二部图，一边是素数，一边是非素数，至于2放在哪里。如果2放到了“素数”那一栏，能得到的最大匹配比放到“非素数”那一栏要更大，那么就放在“素数”那一栏，否则就放到“非素数”那一栏。

这样就建模成了一个二部图。

对于这样的一个二部图，这是一个匹配问题，我们可以将其转换为网络流问题（当然也可以用匈牙利算法），设置一个dummy\_s和一个dummy\_t，我们假设，从dummy\_s->素数->非素数->dummy\_t。那么最大匹配就是这个图的最大流量。设u\_i in 素数，v\_j in “非素数”，<u\_i, v\_j>存在，当且仅当：<u\_i, v\_j>是“素数伴侣“（相加为素数）的时候。



1. import networkx as nx
2. import matplotlib.pyplot as plt
3. from copy import deepcopy
4. from sympy import isprime
5. from collections import deque  *# 双端队列, popleft, append*
6. class solution:
7. def \_\_init\_\_(self, n: int, arr: list[int]) -> None:
8. *# arr就是存放的待匹配的数*
9. self.num = n + 2
10. self.graph: list[list] = [[0] \* self.num for \_ in range(self.num)]
11. *# 只是用作测试*
12. self.arr: list = deepcopy(arr)
13. self.prime: list = [i for i in self.arr if isprime(i)]
14. self.not\_prime: list = [i for i in self.arr if not isprime(i)]
15. self.contain2: bool = 2 in self.arr
16. if self.contain2:
17. self.prime.remove(2)
18. *# 画图*
19. self.G = nx.DiGraph()
20. self.G.add\_node("dummy\_s")  *# 起点*
21. self.G.add\_node("dummy\_t")  *# 终点*
22. for i in range(1, self.num - 1):
23. self.G.add\_node(arr[i - 1])
24. *# dummy\_s = 0; dummy\_s = n + 1*
25. def draw\_graph(self, capa: list, residual: list):
26. len\_prime = len(self.prime)
27. len\_not = len(self.not\_prime)
28. for i in range(len\_prime):
29. for j in range(len\_not):
30. trans\_i = i + 1
31. trans\_j = j + 1 + len\_prime
32. if capa[trans\_i][trans\_j] > 0:
33. self.G.add\_edge(
34. self.prime[i],
35. self.not\_prime[j],
36. weight="{}/{}".format(
37. capa[trans\_i][trans\_j] - residual[trans\_i][trans\_j],
38. capa[trans\_i][trans\_j],
39. ),
40. )
41. for i in range(len\_prime):
42. trans\_i = 1 + i
43. self.G.add\_edge(
44. "dummy\_s",
45. self.prime[i],
46. weight="{}/{}".format(
47. capa[0][trans\_i] - residual[0][trans\_i],
48. capa[0][trans\_i],
49. ),
50. )
51. for j in range(len\_not):
52. trans\_j = 1 + len\_prime + j
53. self.G.add\_edge(
54. self.not\_prime[j],
55. "dummy\_t",
56. weight="{}/{}".format(
57. capa[trans\_j][self.num - 1] - residual[trans\_j][self.num - 1],
58. capa[trans\_j][self.num - 1],
59. ),
60. )
61. pos = nx.shell\_layout(self.G)
62. labels = {n: str(n) for n in self.G.nodes}
63. nx.draw(self.G, pos, with\_labels=True, label=labels)
64. edge\_labels = {(u, v): d["weight"] for u, v, d in self.G.edges(data=True)}
65. nx.draw\_networkx\_edge\_labels(
66. self.G, pos, edge\_labels=edge\_labels, font\_color="red"
67. )
68. plt.title("result")
69. plt.show()
70. def handle(self):
71. if self.contain2:
72. self.prime\_couple(True)
73. capa1 = deepcopy(self.graph)
74. ans1: int = self.FordFulkson(0, self.num - 1)
75. residual1 = deepcopy(self.graph)
76. self.prime.remove(2)
77. self.graph = [[0] \* self.num for \_ in range(self.num)]  *# 重置*
78. self.prime\_couple(False)
79. capa2 = deepcopy(self.graph)
80. ans2: int = self.FordFulkson(0, self.num - 1)
81. residual2 = deepcopy(self.graph)
82. self.not\_prime.remove(2)
83. if ans1 > ans2:
84. self.prime.append(2)
85. self.draw\_graph(capa1, residual1)
86. return ans1
87. else:
88. self.not\_prime.append(2)
89. self.draw\_graph(capa2, residual2)
90. return ans2
91. else:
92. self.prime\_couple(False)
93. capa = deepcopy(self.graph)
94. ans: int = self.FordFulkson(0, self.num - 1)
95. residual = deepcopy(self.graph)
96. self.draw\_graph(capa, residual)
97. return ans
98. def prime\_couple(self, prime\_have\_two: bool):
99. *# 是否要让prime数组中有2*
100. if self.contain2:
101. if prime\_have\_two:  *# 让prime中加入2*
102. self.prime.append(2)
103. len\_prime = len(self.prime)
104. len\_not = len(self.not\_prime)
105. for i in range(len\_prime):
106. for j in range(len\_not):
107. trans\_i = i + 1
108. trans\_j = j + 1 + len\_prime
109. if isprime(self.prime[i] + self.not\_prime[j]):
110. self.graph[trans\_i][trans\_j] = 1
111. else:
112. self.not\_prime.append(2)
113. len\_prime = len(self.prime)
114. len\_not = len(self.not\_prime)
115. for i in range(len\_prime):
116. for j in range(len\_not):
117. trans\_i = i + 1
118. trans\_j = j + 1 + len\_prime
119. if isprime(self.prime[i] + self.not\_prime[j]):
120. self.graph[trans\_i][trans\_j] = 1
121. else:
122. len\_prime = len(self.prime)
123. len\_not = len(self.not\_prime)
124. for i in range(len\_prime):
125. for j in range(len\_not):
126. trans\_i = i + 1
127. trans\_j = j + 1 + len\_prime
128. if isprime(self.prime[i] + self.not\_prime[j]):
129. self.graph[trans\_i][trans\_j] = 1
130. *# 将dummy\_s->"素数"->"非素数“连接起来*
131. for i in range(len(self.prime)):
132. trans\_i = 1 + i
133. self.graph[0][trans\_i] = 1
134. for j in range(len(self.not\_prime)):
135. trans\_j = len(self.prime) + 1 + j
136. self.graph[trans\_j][self.num - 1] = 1
137. def bfs(self, s: int, t: int, pre: list) -> bool:
138. *# 能从s到达t, 那么就返回true*
139. visited = [0] \* self.num
140. m\_q = deque()
141. m\_q.append(s)
142. visited[s] = True
143. while len(m\_q) != 0:
144. u = m\_q.popleft()
145. for v in range(self.num):
146. if not visited[v] and self.graph[u][v] > 0:
147. m\_q.append(v)
148. visited[v] = True
149. pre[v] = u
150. if v == t:
151. return True
152. return False
153. def FordFulkson(self, s, t):
154. pre = [-1] \* self.num
155. max\_flow = 0
156. while self.bfs(s, t, pre):
157. path\_flow = float("inf")  *# 瓶颈*
158. u = t
159. while u != s:
160. path\_flow = min(path\_flow, self.graph[pre[u]][u])
161. u = pre[u]
162. max\_flow += path\_flow
163. v = t
164. while v != s:
165. u = pre[v]
166. self.graph[u][v] -= path\_flow
167. self.graph[v][u] += path\_flow
168. v = pre[v]
169. return max\_flow
170. if \_\_name\_\_ == "\_\_main\_\_":
171. n = 4
172. arr = [2, 5, 6, 13]
173. result = solution(n, arr)
174. ans = result.handle()
175. print(ans)