

ch7_解答

T7.1-2

- T7.1 哈密顿环
- T7.2 八数码

T7.3-4

- T7.3 最短路
- T7.4 哈密顿环（分支界限法）

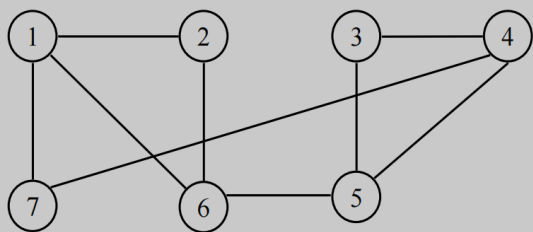
T7.5 子集和

T7.7 组合素数

老师说，搜索问题必考。考题可能不会很灵活（题型有限），可以考虑手玩一下，熟悉操作。

T7.1-2

1.在下图中考虑哈密顿环问题。将问题的解空间表示成树，并分别利用深度优先搜索和广度优先搜索判定该图是否存在哈密顿环。



习题 1

2	3	
1	8	5
7	4	6

起始格局

1	2	3
8		4
7	6	5

目标格局

习题 2

2.考虑 8-魔方问题。分别用深度优先方法，广度优先方法，爬山法，最佳优先方法判定上图所示的初始格局能够通过一系列操作转换成目标格局，将搜索过程的主要步骤书写清楚。


T7.1 哈密顿环

Hamiltonian cycle (BFS): [1, 2, 6, 5, 3, 4, 7, 1]

首先，简要介绍哈密顿环问题：在一个图中，哈密顿环是一条经过所有顶点恰好一次并返回到起点的路径。我们的任务是找到图中是否存在这样的环。

假设我们有以下图的邻接矩阵表示（使用0和1表示边的存在或不存在）：

▼ 邻接矩阵

Plain Text |  复制代码

1

0 1 0 0 0 1 1

2

1 0 0 0 0 1 0

3

0 0 0 1 1 0 0

4

0 0 1 0 1 0 1

5

0 0 1 1 0 1 0

6

1 1 0 0 1 0 0

7

1 0 0 1 0 0 0

1. 深度优先搜索（DFS）：

```

1 def is_valid(v, pos, path):
2     if not adj_matrix[path[pos - 1]][v]:
3         return False
4     if v in path:
5         return False
6     return True
7
8
9 def hamiltonian_cycle_dfs(path, pos):
10     if pos == len(vertices):
11         return adj_matrix[path[pos - 1]][path[0]] == 1
12
13     for v in vertices:
14         if is_valid(v, pos, path):
15             path[pos] = v
16             if hamiltonian_cycle_dfs(path, pos + 1):
17                 return True
18             path[pos] = -1
19
20     return False
21
22
23 vertices = [0, 1, 2, 3, 4, 5, 6]
24 adj_matrix = [
25     [0, 1, 0, 0, 0, 1, 1],
26     [1, 0, 0, 0, 0, 1, 0],
27     [0, 0, 0, 1, 1, 0, 0],
28     [0, 0, 1, 0, 1, 0, 1],
29     [0, 0, 1, 1, 0, 1, 0],
30     [1, 1, 0, 0, 1, 0, 0],
31     [1, 0, 0, 1, 0, 0, 0]
32 ]
33
34 path = [-1] * len(vertices)
35 path[0] = 0
36
37 if hamiltonian_cycle_dfs(path, 1):
38     path.append(path[0])
39     print("Hamiltonian cycle (DFS):", [node+1 for node in path])
40 else:
41     print("No Hamiltonian cycle found (DFS).")

```

2. 广度优先搜索 (BFS) :

```
1  from collections import deque
2
3  def is_valid(v, pos, path):
4      if not adj_matrix[path[pos - 1]][v]:
5          return False
6      if v in path:
7          return False
8      return True
9
10 def hamiltonian_cycle_bfs():
11     q = deque()
12     for v in vertices[1:]:
13         path = [0, v] + [-1] * (len(vertices) - 2)
14         q.append((path, 2))
15
16     while q:
17         path, pos = q.popleft()
18
19         if pos == len(vertices):
20             if adj_matrix[path[-1]][path[0]] == 1:
21                 path.append(path[0])
22                 return path
23             else:
24                 for v in vertices:
25                     if is_valid(v, pos, path):
26                         new_path = path.copy()
27                         new_path[pos] = v
28                         q.append((new_path, pos + 1))
29
30     return None
31
32
33 vertices = [0, 1, 2, 3, 4, 5, 6]
34 adj_matrix = [
35     [0, 1, 0, 0, 0, 1, 1],
36     [1, 0, 0, 0, 0, 1, 0],
37     [0, 0, 0, 1, 1, 0, 0],
38     [0, 0, 1, 0, 1, 0, 1],
39     [0, 0, 1, 1, 0, 1, 0],
40     [1, 1, 0, 0, 1, 0, 0],
41     [1, 0, 0, 1, 0, 0, 0]
42 ]
43
44
45 result = hamiltonian_cycle_bfs()
```

```
46 if result:
47     print("Hamiltonian cycle (BFS):", [node+1 for node in result])
48 else:
49     print("No Hamiltonian cycle found (BFS).")
```

T7.2 八数码

(见仓库里的另一个pdf，有画详细的过程。状态点太多，不想画了)

leetcode简化版八数码：<https://leetcode.cn/problems/sliding-puzzle/>

773. 滑动谜题

难度 1815 301 ☆ 第 69 场周赛 Q3

在一个 2×3 的板上 (board) 有 5 块砖瓦，用数字 1~5 来表示，以及一块空缺用 0 来表示。一次移动定义为选择 0 与一个相邻的数字 (上下左右) 进行交换。

最终当板 board 的结果是 $[[1,2,3],[4,5,0]]$ 谜板被解开。

给出一个谜板的初始状态 board，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1。

示例 1:

1	2	3
4		5

输入: board = $[[1,2,3],[4,0,5]]$

输出: 1

解释: 交换 0 和 5，1 步完成

```

1 class AStar:
2     DIST = [
3         [0, 1, 2, 1, 2, 3],
4         [1, 0, 1, 2, 1, 2],
5         [2, 1, 0, 3, 2, 1],
6         [1, 2, 3, 0, 1, 2],
7         [2, 1, 2, 1, 0, 1],
8         [3, 2, 1, 2, 1, 0],
9     ]
10
11     # 计算启发函数
12     @staticmethod
13     def getH(status: str) -> int:
14         ret = 0
15         for i in range(6):
16             if status[i] != "0":
17                 ret += AStar.DIST[i][int(status[i]) - 1]
18         return ret
19
20     def __init__(self, status: str, g: str) -> None:
21         self.status = status
22         self.g = g
23         self.h = AStar.getH(status)
24         self.f = self.g + self.h
25
26     def __lt__(self, other: "AStar") -> bool:
27         return self.f < other.f
28
29 class Solution:
30     NEIGHBORS = [[1, 3], [0, 2, 4], [1, 5], [0, 4], [1, 3, 5], [2, 4]]
31
32     def slidingPuzzle(self, board: List[List[int]]) -> int:
33         # 枚举 status 通过一次交换操作得到的状态
34         def get(status: str) -> Generator[str, None, None]:
35             s = list(status)
36             x = s.index("0")
37             for y in Solution.NEIGHBORS[x]:
38                 s[x], s[y] = s[y], s[x]
39                 yield "".join(s)
40                 s[x], s[y] = s[y], s[x]
41
42         initial = "".join(str(num) for num in sum(board, []))
43         if initial == "123450":
44             return 0
45

```

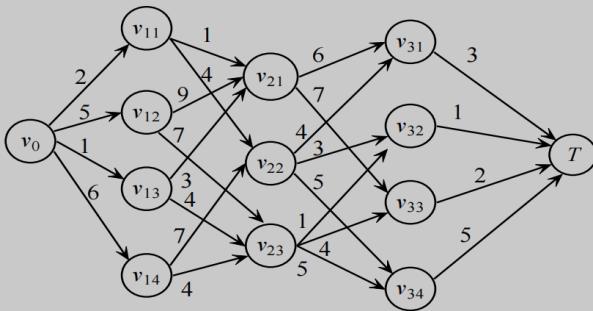
```

46     q = [AStar(initial, 0)]
47     seen = {initial}
48     while q:
49         node = heapq.heappop(q)
50         for next_status in get(node.status):
51             if next_status not in seen:
52                 if next_status == "123450":
53                     return node.g + 1
54             heapq.heappush(q, AStar(next_status, node.g + 1))
55             seen.add(next_status)
56
57     return -1

```

T7.3-4

3. 分别用分支限界法和 A^* 算法求出下图中从 v_0 到 T 的最短路径，写出算法执行的主要过程。



习题 3

	1	2	3	4	5
1	∞	5	61	34	12
2	57	∞	43	20	7
3	39	42	∞	8	21
4	6	50	42	∞	8
5	41	26	10	35	∞

习题 4

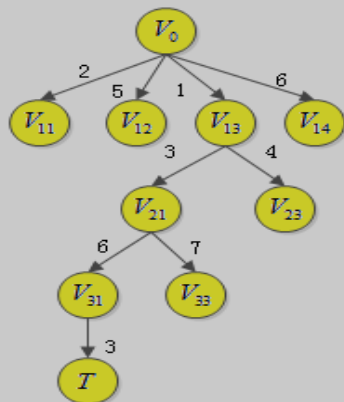
4. 在上面邻接矩阵给出的有向图上，用分支限界法求出代价最小的哈密顿环。

T7.3 最短路

图不想手动输入了，代码就随便拿了一个简单图跑一下

分支界限法:

(1) 利用爬山法贪心的选择当前点所连接的最小边,并按此边进行展开。如下图所示,从 V_0 到 T 的最小路径为: $V_0 - V_{13} - V_{21} - V_{31} - T$,代价为13。

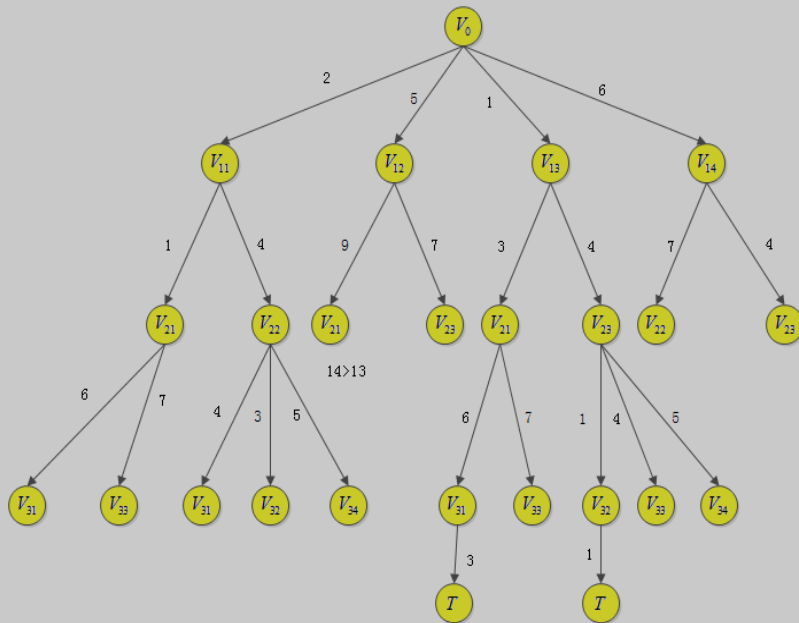


1. 分支限界法:


```
1 import heapq
2
3 def shortest_path_branch_and_bound(graph, start, end):
4     visited = set()
5     queue = [(0, [start])]
6
7     while queue:
8         cost, path = heapq.heappop(queue)
9         current = path[-1]
10
11         if current not in visited:
12             visited.add(current)
13             if current == end:
14                 return cost, path
15
16             for neighbor, distance in enumerate(graph[current]):
17                 if distance > 0 and neighbor not in visited:
18                     heapq.heappush(queue, (cost + distance, path + [neighbor]))
19
20     return None
21
22 graph = [
23     [0, 5, 3, 0, 0, 0],
24     [5, 0, 2, 3, 1, 0],
25     [3, 2, 0, 4, 2, 0],
26     [0, 3, 4, 0, 4, 5],
27     [0, 1, 2, 4, 0, 6],
28     [0, 0, 0, 5, 6, 0]
29 ]
30
31 start, end = 0, 5
32 result = shortest_path_branch_and_bound(graph, start, end)
33 print("Shortest path (Branch and Bound):", result)
```

2. A star

(2) 以代价13为边界，对其他的分支按照爬山法进行展开。如果代价大于13则结束当前路径，如果找到一个最小的路径且代价小于13则对代价重新赋值，重复执行第二步骤，直到所有的点都被遍历,如下图所示，最小路径为：
 $V_0 - V_{13} - V_{23} - V_{32} - T$, 代价为7



我们需要一个启发函数，它将用于A*算法。在这个例子中，我们将使用每个节点到目标节点T的直线距离作为启发函数。

```

1  import heapq
2
3  def heuristic(node, end, heuristics):
4      return heuristics[node]
5
6  def shortest_path_a_star(graph, start, end, heuristics):
7      visited = set()
8      queue = [(0, [start], 0)]
9
10     while queue:
11         f, path, g = heapq.heappop(queue)
12         current = path[-1]
13
14         if current not in visited:
15             visited.add(current)
16             if current == end:
17                 return g, path
18
19             for neighbor, distance in enumerate(graph[current]):
20                 if distance > 0 and neighbor not in visited:
21                     h = heuristic(neighbor, end, heuristics)
22                     heapq.heappush(queue, (g + distance + h, path + [neighbor], g + distance))
23
24     return
25
26  graph = [
27      [0, 5, 3, 0, 0, 0],
28      [5, 0, 2, 3, 1, 0],
29      [3, 2, 0, 4, 2, 0],
30      [0, 3, 4, 0, 4, 5],
31      [0, 1, 2, 4, 0, 6],
32      [0, 0, 0, 5, 6, 0]
33  ]
34
35  heuristics = [10, 4, 6, 2, 4, 0]
36  start, end = 0, 5
37  result = shortest_path_a_star(graph, start, end, heuristics)
38  print("Shortest path (A *):", result)

```

T7.4 哈密顿环（分支界限法）

$$\begin{array}{c}
 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
 \begin{pmatrix}
 \infty & 5 & 61 & 34 & 12 \\
 57 & \infty & 43 & 20 & 7 \\
 39 & 42 & \infty & 8 & 21 \\
 6 & 50 & 42 & \infty & 8 \\
 41 & 26 & 10 & 35 & \infty
 \end{pmatrix}
 \end{array}$$

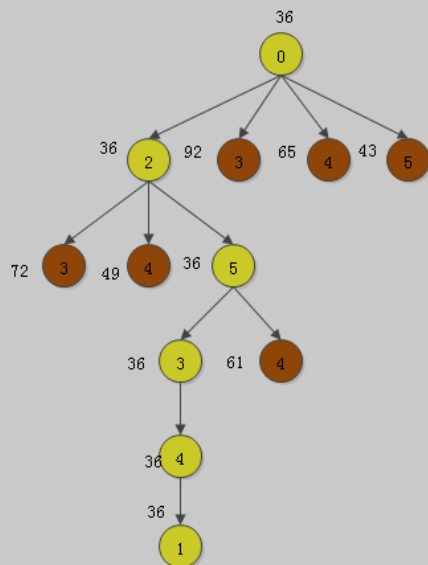
解:

(1) 将代价矩阵的每行(列)减去同一个数(该行或列的最小数),使得每行和每列至少有一个零,其余各元素非负。

$$\begin{bmatrix} \infty & 5 & 61 & 34 & 12 \\ 57 & \infty & 43 & 20 & 7 \\ 39 & 42 & \infty & 8 & 21 \\ 6 & 50 & 42 & \infty & 8 \\ 41 & 26 & 10 & 35 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & 0 & 56 & 29 & 7 \\ 50 & \infty & 36 & 13 & 0 \\ 31 & 34 & \infty & 0 & 13 \\ 0 & 44 & 36 & \infty & 2 \\ 31 & 16 & 0 & 25 & \infty \end{bmatrix}$$

经过变换后解的下界为: $5+7+8+6+10=36$

(2) 解空间的加权树:



代价最小的哈密顿环为: $v_1 - v_2 - v_5 - v_3 - v_4 - v_1$

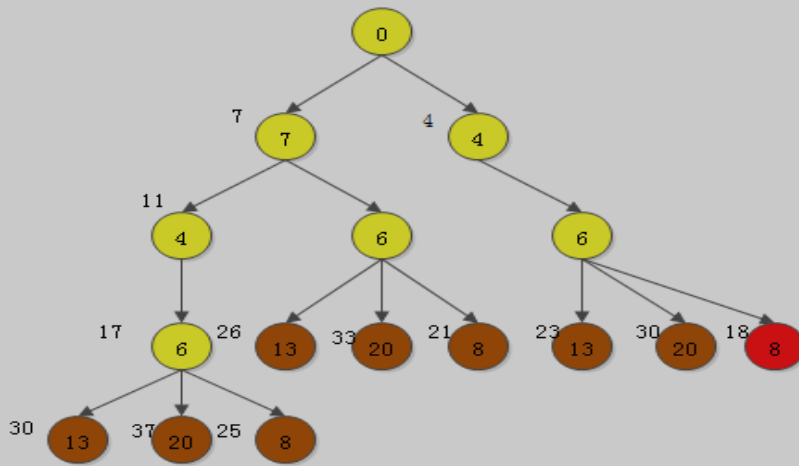
T7.5 子集和

5. 分别使用深度优先法和分支限界法求解子集和问题的如下实例。

输入：集合 $S=\{7,4,6,13,20,8\}$ 和整数 $K=18$

输出： $S'\subseteq S$ 使得 S' 中元素之和等于 K

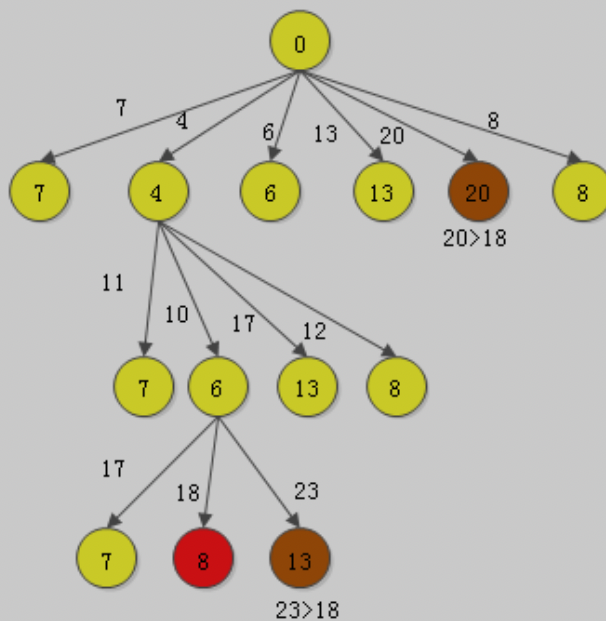
深度优先搜索：



分支界限法：

(1) 确定问题的上界为 18

(2) 利用爬山法进行问题搜索，每次选择当前可扩展的最小元素，并利用上界进行减枝。



子集和问题是一个NP完全问题，因此没有已知的多项式时间解决方案。但是，我们可以使用深度优先搜索（DFS）和分支限界法（Branch and Bound）来找到满足条件的子集。以下是针对给定问题实例的两种算法的Python实现：

1. 深度优先搜索（DFS）：

```
1 def subset_sum_dfs(arr, k, path, start):
2     if k == 0:
3         return path
4
5     for i in range(start, len(arr)):
6         remaining = k - arr[i]
7         if remaining >= 0:
8             path.append(arr[i])
9             result = subset_sum_dfs(arr, remaining, path, i + 1)
10            if result:
11                return result
12            path.pop()
13
14    return None
15
16 arr = [7, 4, 6, 13, 20, 8]
17 k = 18
18
19 result = subset_sum_dfs(arr, k, [], 0)
20 if result:
21     print("Subset found (DFS):", result)
22 else:
23     print("No subset found (DFS).")
```

1. 分支限界法 (Branch and Bound) :

```
1 def subset_sum_bnb(arr, k):
2     n = len(arr)
3     arr.sort(reverse=True)
4     upper_bound = [0] * (n + 1)
5     upper_bound[-1] = 0
6
7     for i in range(n - 1, -1, -1):
8         upper_bound[i] = upper_bound[i + 1] + arr[i]
9
10    def bnb(start, k, path):
11        if k == 0:
12            return path
13
14        if start < n and upper_bound[start] >= k:
15            remaining = k - arr[start]
16            if remaining >= 0:
17                path.append(arr[start])
18                result = bnb(start + 1, remaining, path)
19                if result:
20                    return result
21                path.pop()
22
23            result = bnb(start + 1, k, path)
24            if result:
25                return result
26
27        return None
28
29    return bnb(0, k, [])
30
31 arr = [7, 4, 6, 13, 20, 8]
32 k = 18
33
34 result = subset_sum_bnb(arr, k)
35 if result:
36     print("Subset found (Branch and Bound):", result)
37 else:
38     print("No subset found (Branch and Bound).")
```

这两个算法都接受一个数组arr和一个整数k，寻找一个子集，使得子集中的元素之和等于k。第一个算法使用深度优先搜索，而第二个算法使用分支限界法。在这个问题实例中，两种方法都可以找到满足条件的子集[4, 6, 8]。

T7.7 组合素数

7. 选用恰当的搜索策略，求解如下计算问题。

输入： 正整数集合 $A=\{a_1, a_2, \dots, a_n\}$ 和正整数 K

输出： 由 A 中任意 K 个数相关得到的最小素数

(1) 将问题的解空间表示成一棵树；

(2) 写出求解问题的通用算法；

(3) 在问题实例 $A=\{3, 7, 12, 19\}$ ， $K=3$ 上写出算法运行的主要过程。

这个问题可以使用dfs来解决。我们的目标是找到由给定集合A中任意K个数字组成的最小素数。

1. 将问题的解空间表示成一棵树：

解空间可以表示为一棵树，其中根节点表示我们尚未选择任何数字，每个节点的子节点表示从剩余数字中选择一个数字。树的深度限制为K。树的叶子节点表示已选择K个数字的所有可能组合。

2. 通用算法：


```

1  from itertools import permutations
2  import math
3
4  def is_prime(n):
5      if n <= 1:
6          return False
7      if n <= 3:
8          return True
9      if n % 2 == 0 or n % 3 == 0:
10         return False
11     i = 5
12     while i * i <= n:
13         if n % i == 0 or n % (i + 2) == 0:
14             return False
15         i += 6
16     return True
17
18 def min_prime_from_k_numbers(A, K):
19     min_prime = float('inf')
20
21     for p in permutations(A, K): #库里实现了枚举搜索
22         number = int(''.join(map(str, p)))
23         if is_prime(number) and number < min_prime:
24             min_prime = number
25
26     return min_prime if min_prime != float('inf') else None
27
28 A = [3, 7, 12, 19]
29 K = 3
30
31 result = min_prime_from_k_numbers(A, K)
32 if result:
33     print("Minimum prime from", K, "numbers in A:", result)
34 else:
35     print("No prime number can be formed from", K, "numbers in A.")

```

3. 主要过程:

给定集合A={3,7,12,19}和K=3，算法首先生成A中所有长度为K的排列组合。在这个例子中，有4个数字和3个数字需要选择，所以总共有 $4!/(4-3)! = 24$ 种可能的组合。接下来，算法会将每种组合转换为整数，然后检查其是否为素数。如果找到素数且其值小于当前的最小素数，则更新最小素数。

在这个例子中，最小的素数是1237。