38 猫和老鼠

更新时间：2019-09-30 09:45:20

> 天才免不了有障碍，因为障碍会创造天才。
>
> ——罗曼·罗兰

刷题内容

> **难度: Hard**
>
> 题目链接：https://leetcode-cn.com/problems/cat-and-mouse/

题目描述

两个玩家分别扮演猫（Cat）和老鼠（Mouse）在无向图上进行游戏，他们轮流行动。

该图按下述规则给出：graph[a] 是所有结点 b 的列表，使得 ab 是图的一条边。

老鼠从结点 1 开始并率先出发，猫从结点 2 开始且随后出发，在结点 0 处有一个洞。

在每个玩家的回合中，他们必须沿着与他们所在位置相吻合的图的一条边移动。例如，如果老鼠位于结点 1，那么它只能移动到 graph[1] 中的（任何）结点去。

此外，猫无法移动到洞（结点 0）里。

然后，游戏在出现以下三种情形之一时结束：

如果猫和老鼠占据相同的结点，猫获胜。
如果老鼠躲入洞里，老鼠获胜。
如果某一位置重复出现（即，玩家们的位置和移动顺序都与上一个回合相同），游戏平局。
给定 graph，并假设两个玩家都以最佳状态参与游戏，如果老鼠获胜，则返回 1；如果猫获胜，则返回 2；如果平局，则返回 0。

示例：

输入：[[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
输出：0
解释：
4---3---1
| |
2---5
 \ /
  0

提示：

3 <= graph.length <= 200
保证 graph[1] 非空。
保证 graph[2] 包含非零元素。

解题方案

思路1 时间复杂度: O(N^3) 空间复杂度: O(N^2)

不绕弯子，这道题是零和博弈的题，是一类典型题目。

令f(turn, mouse, cat)表示当老鼠的位置在mouse，猫的位置在cat，并且turn为目前的回合是猫行动还是老鼠行动（turn=1 老鼠行动 ；turn=2猫行动），经过一系列行动后，是猫赢还是老鼠赢。函数值等于1时，老鼠赢，函数值等于2时，猫赢，函数值等于0时，平局。

显然，总状态数是o(2 * n * n)。

我们站在老鼠的角度想问题，如果轮到老鼠走，老鼠肯定会走到一个有利于自己的状态，假设后面有一个状态，这个状态会导致老鼠赢，那么老鼠肯定坚定的往这个状态移动，能赢。如果后面所有的状态，都会导致老鼠输，那么老鼠绝对不可能赢，也不可能平局。除此之外，如果后续没有一个赢的状态，且至少出现一个平局的状态，那么老鼠就会转移到平局的状态。

猫也是这么想的。

题目目标求f(1, 1, 2)

我们可以倒过来求f(1,1,2)，首先，我们先来看边界条件。

- 猫和老鼠在同一个位置，猫赢
- 老鼠在0这个位置，老鼠赢
- 猫不可以出现在0

我们先把边界条件给计算好，用一个队列，来保存。对于每一个猫赢或者老鼠赢的状态A，遍历这个状态的上一个状态B，那么A就是B的下一个状态。我们检查状态B，如果状态B是老鼠先走，检查状态B的所有的后继状态，如果存在一个状态是老鼠赢，那么状态B也是老鼠赢。如果所有后继状态都是猫赢，那么状态B就是猫赢。除此之外，***暂定***为平局。如果状态B是老鼠赢或者猫赢，该状态进入队列以待后续展开。

优化点：我们可以为每个状态保存一个后继状态可以必输多少个。显然这个值初始等于后继状态的数量。当我们计算出状态A，如果是必赢的状态，那么状态B就是必赢，如果状态A是必输，那么把可以必输的数量减1，直到把必输的数量减到1，那么状态B就是必输。

这样优化后。时间复杂度是o(n^3 * cache的复杂度)

Python

beats 39.8%

```python
class Solution:
    def catMouseGame(self, graph: List[List[int]]) -> int:
        def prev(graph, nxt): # 求前驱状态
            res = []
            if nxt[0] == 2:
                for x in graph[nxt[1]]:
                    res.append([1, x, nxt[2]])
            else:
                for x in graph[nxt[2]]:
                    if x == 0:
                        continue
                    res.append([2, nxt[1], x])
            return res

        cache = {} # 1老鼠赢，2猫赢
        degree = {} # 可以必输的数量

        for i in range(len(graph)):
            for j in range(1, len(graph)):
                degree[tuple([1,i,j])] = len(graph[i])
                cnt = 0 # 由于猫不能到0，所以要重新统计下
                for x in graph[j]:
                    if x != 0:
                        cnt += 1
                degree[tuple([2,i,j])] = cnt

        queue = collections.deque()
        for i in range(1, len(graph)):
            # 猫和老鼠在一个地方
            cache[tuple([1,i,i])] = 2
            queue.append(tuple([1,i,i]))
            cache[tuple([2,i,i])] = 2
            queue.append(tuple([2,i,i]))
            # 老鼠在0
            cache[tuple([1,0,i])] = 1
            queue.append(tuple([1,0,i]))
            cache[tuple([2,0,i])] = 1
            queue.append(tuple([2,0,i]))

        while queue:
            x = queue.popleft()
            for pre in prev(graph, x):
                pre = tuple(pre)
                if pre in cache:
                    continue
                # 必赢的状态
                if cache[x] == pre[0]:
                    cache[pre] = pre[0]
                    queue.append(pre)
                else:
                    degree[pre] -= 1
                    if degree[pre] == 0:
                        # 必输的状态
                        cache[pre] = 3 - pre[0]
                        queue.append(pre)
        return cache.get(tuple([1,1,2]), 0)
```

Golang

beats 33.33%

```golang
func prev(graph [][]int, nxt []int) [][]int {
    res := [][]int{}
    if nxt[0] == 2 {
        for _, x := range graph[nxt[1]] {
            res = append(res, []int{1, x, nxt[2]})
        }
```

```
        }
    } else {
        for _, x := range graph[nxt[2]] {
            if x == 0 {
                continue
            }
            res = append(res, []int{2, nxt[1], x})
        }
    }
    return res
}

// 将值放入map中
func setMap(cache map[int]map[int]map[int]int, i, j, k int, v int) {
    if _, ok1 := cache[i]; !ok1 {
        cache[i] = map[int]map[int]int{}
    }
    if _, ok1 := cache[i][j]; !ok1 {
        cache[i][j] = map[int]int{}
    }
    cache[i][j][k] = v
}

// 看值是否在map中
func inMap(cache map[int]map[int]map[int]int, i, j, k int) bool {
    if _, ok1 := cache[i]; ok1 {
        if _, ok2 := cache[i][j]; ok2 {
            if _, ok3 := cache[i][j][k]; ok3 {
                return true
            }
        }
    }
    return false
}

func catMouseGame(graph [][]int) int {
    cache := map[int]map[int]map[int]int{} //1老鼠赢，2猫赢
    degree := map[int]map[int]map[int]int{} //可以必输的数量
    for i := 0; i < len(graph); i++ {
        for j:= 1; j < len(graph); j++ {
            setMap(degree, 1, i, j, len(graph[i]))
            cnt := 0 //由于猫不能到0，所以要重新统计下
            for _, x := range graph[j] {
                if x != 0 {
                    cnt += 1
                }
            }
            setMap(degree, 2, i, j, cnt)
        }
    }

    queue := [][]int{}
    for i := 1; i < len(graph); i++ { //猫不能进洞，从1开始遍历
        //猫和老鼠在一个地方
        setMap(cache, 1, i, i, 2)
        queue = append(queue, []int{1,i,i})
        setMap(cache, 2, i, i, 2)
        queue = append(queue, []int{2,i,i})
        //老鼠在0
        setMap(cache, 1, 0, i, 1)
        queue = append(queue, []int{1,0,i})
        setMap(cache, 2, 0, i, 1)
        queue = append(queue, []int{2,0,i})
    }

    for len(queue) > 0 {
        x := queue[0]
        queue = queue[1:]
        for _, pre := range prev(graph, x) {
            if inMap(cache, pre[0], pre[1], pre[2]) {
```

```
            if inMap(cache, pre[0], pre[1], pre[2]) {
                continue
            }
            //必赢的状态
            if cache[x[0]][x[1]][x[2]] == pre[0] {
                setMap(cache, pre[0], pre[1], pre[2], pre[0])
                queue = append(queue, pre)
            } else {
                degree[pre[0]][pre[1]][pre[2]] -= 1
                if degree[pre[0]][pre[1]][pre[2]] == 0 {
                    //必输的状态
                    setMap(cache, pre[0], pre[1], pre[2], 3 - pre[0])
                    queue = append(queue, pre)
                }
            }
        }
    }

    if _, ok1 := cache[1]; ok1 {
        if _, ok2 := cache[1][1]; ok2 {
            if _, ok3 := cache[1][1][2]; ok3 {
                return cache[1][1][2]
            }
        }
    }
    return 0
}
```

Java

beats 6.28%

```
import java.util.LinkedList;
import java.util.Queue;

class Solution {
    private class Node {
        int first;
        int mouse;
        int cat;
        // 声明Node
        public Node(int first, int cat, int mouse) {
            this.first = first;
            this.mouse = mouse;
            this.cat = cat;
        }

        public int getFirst() {
            return first;
        }

        public void setFirst(int first) {
            this.first = first;
        }

        public int getMouse() {
            return mouse;
        }

        public void setMouse(int mouse) {
            this.mouse = mouse;
        }

        public int getCat() {
            return cat;
        }
```

```
    public void setCat(int cat) {
        this.cat = cat;
    }
}

private int solve(int[][] graph) {
    int[][][] cache = new int[2][graph.length][graph.length];
    for (int first = 0;first < 2;first++) {
        for (int cat = 0;cat < graph.length;cat++) {
            for (int mouse = 0;mouse < graph.length;mouse++) {
                cache[first][cat][mouse] = 0;
            }
        }
    }

    Queue<Node> nodes = new LinkedList<>();

    for (int first = 0;first < 2;first++) {
        for (int cat = 1; cat < graph.length; cat++) {
            for (int mouse = 0; mouse < graph.length; mouse++) {
                if (mouse == 0) { // 老鼠在0
                    cache[first][cat][mouse] = 1;
                    nodes.add(new Node(first, cat, mouse));
                }
                if (cat == mouse) { // 猫和老鼠在一个地方
                    cache[first][cat][mouse] = 2;
                    nodes.add(new Node(first, cat, mouse));
                }
            }
        }
    }

    while (!nodes.isEmpty()) { // 只要nodes非空
        Node node = nodes.poll();

        if (node.first == 0) {
            for (int i = 0;i < graph[node.mouse].length;i++) {
                int x = graph[node.mouse][i];

                int pre = cache[1][node.cat][x];
                if (x == 0 || node.cat == x) {
                    continue;
                }

                boolean findWin = false;
                boolean findDraw = false;
                for (int j = 0;j < graph[x].length;j++) {
                    int y = graph[x][j];
                    if (cache[0][node.cat][y] == 1) {
                        findWin = true;
                    } else if (cache[0][node.cat][y] == 0) {
                        findDraw = true;
                    }
                }
                if (findWin) {
                    cache[1][node.cat][x] = 1;
                } else if (!findDraw) {
                    cache[1][node.cat][x] = 2;
                } else {
                    cache[1][node.cat][x] = 0;
                }

                if (cache[1][node.cat][x] != pre) {
                    nodes.add(new Node(1, node.cat, x));
                }
            }
        } else {
            for (int i = 0;i < graph[node.cat].length;i++) {
                int x = graph[node.cat][i];
```

```
                int pre = cache[0][x][node.mouse];

                if (x == 0 || node.mouse == x) {
                    continue;
                }

                boolean findWin = false;
                boolean findDraw = false;
                for (int j = 0;j < graph[x].length;j++) {
                    int y = graph[x][j];
                    if (y == 0) {
                        continue;
                    }
                    if (cache[1][y][node.mouse] == 2) {
                        findWin = true;
                    } else if (cache[1][y][node.mouse] == 0) {
                        findDraw = true;
                    }
                }
                if (findWin) {
                    cache[0][x][node.mouse] = 2;
                } else if (!findDraw) {
                    cache[0][x][node.mouse] = 1;
                } else {
                    cache[0][x][node.mouse] = 0;
                }

                if (pre != cache[0][x][node.mouse]) {
                    nodes.add(new Node(0, x, node.mouse));
                }
            }
        }
    }

    return cache[1][2][1];
  }

  public int catMouseGame(int[][] graph) {
    int ret = solve(graph);
    return ret;
  }
}
```

c++

beats 5%

```
class Solution {
public:
  //求前驱状态
  vector<vector<int>> _prev(vector<vector<int>>& graph, vector<int>& next) {
    vector<vector<int>> ret;
    if (next[0] == 2) {
      for (auto x : graph[next[1]]) {
        ret.push_back({1, x, next[2]});
      }
    } else {
      for (auto x : graph[next[2]]) {
        if (x == 0) continue;
        ret.push_back({2, next[1], x});
      }
    }
    return ret;
  }
  int catMouseGame(vector<vector<int>>& graph) {
    //1老鼠赢，2猫赢
```

```
        map<vector<int>, int> cache;
        //可以必输的数量
        map<vector<int>, int> degree;
        for (int i = 0;i < graph.size();i++) {
            for (int j = 1;j < graph.size();j++) {
                degree[{1,i,j}] = graph[i].size();
                //由于猫不能到0，所以要重新统计下
                int count = 0;
                for (auto x: graph[j]) {
                    if (x != 0) {
                        count++;
                    }
                }
                degree[{2,i,j}] = count;
            }
        }
        queue<vector<int>> Q;
        //猫不能进洞，从1开始遍历
        for(int i = 1;i < graph.size();i++) {
            //猫和老鼠在一个地方
            cache[{1,i,i}] = 2;
            Q.push({1,i,i});
            cache[{2,i,i}] = 2;
            Q.push({2,i,i});
            //老鼠在0
            cache[{1,0,i}] = 1;
            Q.push({1,0,i});
            cache[{2,0,i}] = 1;
            Q.push({2,0,i});
        }

        while (!Q.empty()) {
            vector<int> x = Q.front();
            Q.pop();

            for (auto prev: _prev(graph, x)) {
                if (cache.find(prev) != cache.end()) {
                    continue;
                }

                //必赢的状态
                if (cache[x] == prev[0]) {
                    cache[prev] = prev[0];
                    Q.push(prev);
                } else {
                    degree[prev]--;
                    if (degree[prev] == 0) {
                        //必输的状态
                        cache[prev] = 3 - prev[0];
                        Q.push(prev);
                    }
                }
            }
        }
        return cache[{1, 1, 2}];
    }
};
```

总结

零和博弈是一类经典题目。由于博弈双方都是对自己是最优策略，如果后序状态出现必赢状态，那么自己肯定会走到该状态。

这一类题目的关键在于找准状态的转移和变化，注意边界输赢的情况，利用递推或者记忆化搜索来做状态的计算。

经典题目：有一堆石头共有n个，AB两个人轮流拿石子，每次最多拿x个，最先拿完最后一个石子的人胜利。A先拿，假设两个人都用最佳策略来进行游戏，问A能否胜出。

这题就留给大家思考

}

37 和最少为K的最短子数组

39 专题1：LRU Cache 最近最少使用算法