

2023-9-5 狙击

- 1. 参数解释
 - 2. 初始化 `dp[u]`
 - 3. 遍历 `u` 的所有邻边
 - 4. 递归调用 `dfs(v, u)`
 - 5. 更新 `dp[u][1]` 和 `dp[u][2]`
 - 6. 更新 `dp[u][0]`
- 为什么要用 `dp[u][2]` -- 次长边

dfs - 树形dpPython

```
1 def dfs(u, fa):
2     dp[u][0] = dp[u][1] = dp[u][2] = 0
3     for idx in graph[u]:
4         edge = e[idx]
5         v = edge.u ^ edge.v ^ u
6         if v == fa:
7             continue
8         w = edge.b - edge.w
9         dfs(v, u)
10
11         if dp[v][1] + w > dp[u][1]:
12             dp[u][2] = dp[u][1]
13             dp[u][1] = dp[v][1] + w
14         elif dp[v][1] + w > dp[u][2]:
15             dp[u][2] = dp[v][1] + w
16
17     dp[u][0] = max(dp[u][0], dp[v][0], dp[u][1] + dp[u][2])
```

1. 参数解释

- `u` : 当前遍历的节点。
- `fa` : `u` 的父节点，用于避免回到父节点，造成死循环。

2. 初始化 `dp[u]`

`dp[u]` 是动态规划数组，定义如下：

- `dp[u][0]` : 表示以 `u` 为根的子树中可以形成的最大路径值。
- `dp[u][1]` 和 `dp[u][2]` : 分别表示以 `u` 为根的子树中可以向下延伸的最大和次大路径值。

初始时，`dp[u]` 被重置为 `[0, 0, 0]`，表示从 `u` 出发还未计算任何路径。

3. 遍历 `u` 的所有邻边

`graph[u]` 存储了与 `u` 相连的边的索引，遍历这些边时：

- `edge.u ^ edge.v ^ u` 通过异或运算获取当前边的另一个端点 `v`。
- 如果 `v` 是父节点 `fa`，说明 `v` 是从 `u` 回去的节点，跳过，避免重复遍历。
- 计算 `w = edge.b - edge.w`，表示当前边的权重差值（或路径值）。

4. 递归调用 `dfs(v, u)`

对节点 `v` 进行递归调用，继续深度优先搜索其子节点。递归返回后，`dp[v]` 存储了 `v` 相关的路径信息。

5. 更新 `dp[u][1]` 和 `dp[u][2]`

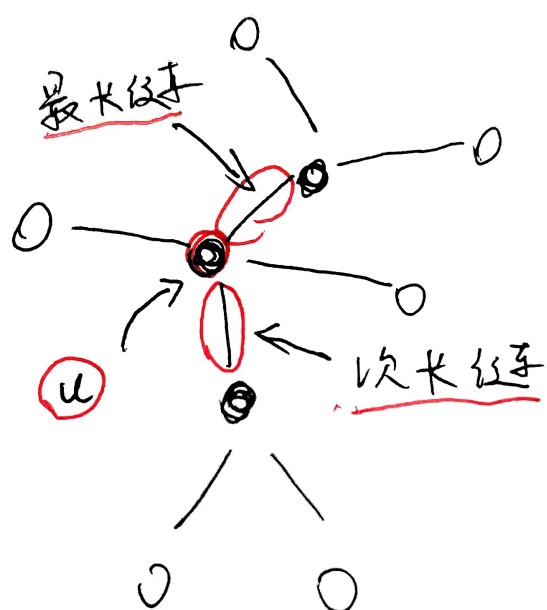
- 判断 `dp[v][1] + w` 是否大于 `dp[u][1]`（当前最大路径值），如果是，则更新 `dp[u][1]` 和 `dp[u][2]`（最大和次大路径值）。
- 否则，判断是否大于 `dp[u][2]`，如果是，则更新 `dp[u][2]`（次大路径值）。

6. 更新 `dp[u][0]`

- `dp[u][0]` 的值是当前节点 `u` 的最大路径值、子节点 `v` 的最大路径值，以及通过 `dp[u][1] + dp[u][2]` 形成的路径中的最大值。

为什么要用 `dp[u][2]` -- 次长边

如下图，对于当前节点 `u`，可能图中的最长链包含了 `u`，但实际上 `u` 不是端点。



```
1  # 超时 8分
2  # 超时 8分
3  # 超时 8分
4  # 超时 8分
5  # 超时 8分
6  # 超时 8分
7  # 超时 8分
8  # 超时 8分
9  # 超时 8分
10 # 超时 8分
11 # 超时 8分
12 # 超时 8分
13 # 超时 8分
14 # 超时 8分
15
16 import sys
17 from collections import defaultdict
18
19 sys.setrecursionlimit(200000)
20
21 N = 200005
22 mod = 998244353
23
24 class Edge:
25     def __init__(self, u, v, w, b):
26         self.u = u
27         self.v = v
28         self.w = w
29         self.b = b
30
31 graph = defaultdict(list)
32 e = [None] * N
33 dp = [[0] * 3 for _ in range(N)]
34
35 def dfs(u, fa):
36     dp[u][0] = dp[u][1] = dp[u][2] = 0
37     for idx in graph[u]:
38         edge = e[idx]
39         v = edge.u ^ edge.v ^ u
40         if v == fa:
41             continue
42         w = edge.b - edge.w
43         dfs(v, u)
44
45         if dp[v][1] + w > dp[u][1]:
46             dp[u][2] = dp[u][1]
47             dp[u][1] = dp[v][1] + w
48         elif dp[v][1] + w > dp[u][2]:
49             dp[u][2] = dp[v][1] + w
50
51     dp[u][0] = max(dp[u][0], dp[v][0], dp[u][1] + dp[u][2])
52
53 def work():
54     n, m = map(int, input().split())
```

```

55     tot = 0
56
57     for _ in range(n - 1):
58         tot += 1
59         u, v, w, b = map(int, input().split())
60         e[tot] = Edge(u, v, w, b)
61         graph[u].append(tot)
62         graph[v].append(tot)
63
64     dfs(1, -1)
65     print(dp[1][0])
66
67     for _ in range(m):
68         x, y = map(int, input().split())
69         e[x].w = y
70         dfs(1, -1)
71         print(dp[1][0])
72
73     if __name__ == "__main__":
74         # Reading input directly from the standard input
75         work()
76
77     '''
78     5 3
79     1 2 6 4
80     2 3 2 1
81     3 4 5 3
82     3 5 8 5
83     3 2
84     4 3
85     1 1
86
87     '''

```