慧灯 vs 传统大模型

输入同一个问题，实时对比两边的回答、成本、速度与质量

在社交网络，迷宫或者树状结构图上进行深度优先搜索到一个目标的一种遍历方法，写代码

开始对比

试试： 决策分析 深度分析 创作生成 代码执行 事实核查 偏见测试 幻觉测试

成本节省	响应省时	幻觉检测	偏见检测
85.6%	77.8%（4.5倍速）	无	无

对比分析

成本对比：慧灯消耗 **¥0.000930**，传统模型消耗 **¥0.006483**，慧灯节省 **85.6%**。

思考链成本：传统模型有 **87%** 的输出 Token 花在思考链上，对应成本 **¥0.005640**——用户看不见，却要为此付费。

速度对比：慧灯用时 **5.89s**，传统模型用时 **26.57s**，慧灯省时 **77.8%**（响应速度为传统模型的 **4.5 倍**）。

Token 对比：慧灯输出 794 tokens（思考链 0），传统模型输出 5879 tokens（其中思考链 5115）。

质量对比（逐维度直接读两边原文）：

· 前瞻性**慧灯更优**——慧灯明确指出「若不做 visited 标记，社交图（有环）会无限递归或栈溢出」及递归深度风险，并给出验证信号，传统模型只泛泛说明 visited 防死循环。

· 智慧**慧灯更优**——慧灯开头直陈本质「不是找一条路径，而是带访问标记的 DFS 找到目标即回溯」，并点明「要最短路径别用 DFS」，传统模型说明较平铺啰嗦。

· 幻觉：慧灯无，传统模型无。两份回答的代码与示例均可运行，未出现无依据的数字、人名或奖项等具体断言。

· 偏见：慧灯无，传统模型无。两份回答均未含刻板印象或不当偏向。

模型： [redacted] · 使用你的 Key

慧灯

5.89s · ¥0.000930

不是“找一条路径”，而是“在带环的社交/迷宫图上做有访问标记的 DFS，找到目标即回溯出路径”。

趋势预判：若不做 visited 标记，社交图（有环）会无限递归或栈溢出；若用递归且深度过万，Python 默认递归上限会先崩。验证信号：跑一个含环的图，看是否重复访问同一节点。

现在能做的：用显式栈 + visited 的迭代 DFS，同时记录 parent 以还原路径。

```
def dfs_path(graph, start, goal):
    """
```

传统模型（开思考）

26.57s · ¥0.006483

可以把社交网络、树、迷宫都抽象成“图”：每个状态是一个节点，用 `neighbors(node)` 返回它的相邻节点。下面是一个通用 DFS，能找到从 `start` 到 `target` 的路径，并返回访问顺序。

```
def dfs_find_path(start, target, neighbors):
    """
    通用深度优先搜索 DFS
    start: 起点
    target: 目标节点
    neighbors: 函数，neighbors(node) 返回 node 的所有邻居
    返回: (路径 list, 访问顺序 list)，找不到路径则路径为
```

```
    graph: dict[node, list[node]] 邻接表（社交网络/迷宫/树通用）
    返回: 从 start 到 goal 的一条路径列表；不存在返回 None
    """
    if start == goal:
        return [start]

    visited = {start}
    parent = {start: None}
    stack = [start]

    while stack:
        node = stack.pop()
        for nxt in graph.get(node, []):
            if nxt in visited:
                continue
            visited.add(nxt)
            parent[nxt] = node
            if nxt == goal:
                # 回溯路径
                path = []
                cur = goal
                while cur is not None:
                    path.append(cur)
                    cur = parent[cur]
                return path[::-1]
            stack.append(nxt)
    return None


---- 示例 ----

if __name__ == "__main__":
    # 社交网络：有环
    social = {
        "A": ["B", "C"],
        "B": ["A", "D", "E"],
        "C": ["A", "F"],
        "D": ["B"],
        "E": ["B", "F"],
        "F": ["C", "E", "G"],
        "G": ["F"],
    }
    print(dfs_path(social, "A", "G"))  # ['A', 'C', 'F', 'G'] 或 ['A','B','E','F','G']

    # 迷宫：网格转邻接表
    maze = [[0,0,0],
            [1,1,0],
            [0,0,0]]
```

```
None
    """
    stack = [start]
    visited = {start}
    parent = {start: None}  # 记录路径前驱
    visit_order = []        # 记录访问顺序

    while stack:
        node = stack.pop()
        visit_order.append(node)

        if node == target:
            # 回溯路径
            path = []
            cur = node
            while cur is not None:
                path.append(cur)
                cur = parent[cur]
            path.reverse()
            return path, visit_order

        for nxt in neighbors(node):
            if nxt not in visited:
                visited.add(nxt)
                parent[nxt] = node
                stack.append(nxt)

    return None, visit_order


==========


1. 社交网络 / 树：使用邻接表


==========


social_graph = {
    "我": ["张三", "李四"],
    "张三": ["王五"],
    "李四": ["王五", "赵六"],
    "王五": ["目标用户"],
    "赵六": [],
    "目标用户": []
}

path, order = dfs_find_path(
    "我",
    "目标用户",
```

```
    def build(maze):
        R, C = len(maze), len(maze[0])
        g = {}
        for r in range(R):
            for c in range(C):
                if maze[r][c] == 1:
                    continue
                g[(r,c)] = []
                for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
                    nr, nc = r+dr, c+dc
                    if 0 <= nr < R and 0 <= nc < C and
maze[nr][nc] == 0:
                        g[(r,c)].append((nr,nc))
        return g
    print(dfs_path(build(maze), (0,0), (2,2)))  # [(0,0),
(0,1),(0,2),(1,2),(2,2)]
```

边界

- `start == goal` 直接返回单点路径。
- 目标不可达返回 `None`。
- 图可为不连通；`graph.get(node, [])` 兼容孤立节点。
- 自环/重边由 `visited` 自动屏蔽。

复杂度

- 时间 O(V + E)，每个节点、每条边最多访问一次。
- 空间 O(V)，`visited`/`parent`/`stack` 均为线性。

一条判断：要“最短路径”别用 DFS，换 BFS——DFS 只保证“存在一条路径”，不保证最短。

Prompt Tokens	**211**
Completion Tokens	**794**
思考链 Tokens	**0**
max_tokens	**8000**
思考模式	**关闭**

```
    lambda node: social_graph.get(node, [])
)

print("社交网络访问顺序：", order)
print("社交网络路径：", path)
```

==========

2. 迷宫：S 起点，T 目标，# 墙，. 通路

==========

```
def solve_maze(maze):
    rows, cols = len(maze), len(maze[0])
    start = target = None

    for r in range(rows):
        for c in range(cols):
            if maze[r][c] == "S":
                start = (r, c)
            elif maze[r][c] == "T":
                target = (r, c)

    def neighbors(pos):
        r, c = pos
        # 上下左右四个方向
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and
maze[nr][nc] != "#":
                yield (nr, nc)

    return dfs_find_path(start, target, neighbors)


maze = [
    "S..#",
    ".#..",
    ".#T.",
]

maze_path, maze_order = solve_maze(maze)
print("迷宫访问顺序：", maze_order)
print("迷宫路径：", maze_path)
```

说明：

• `stack` 是 DFS 的核心，后进先出。

• `visited` 防止重复访问，避免死循环。

• `parent` 用来记录每个节点是从哪里来的，从而在找到目标后回溯出完整路径。

• DFS 找到的路径不一定是最短路径；如果需要最短路径，可以把 `stack` 换成 `queue`，使用 BFS。

Prompt Tokens	**59**
Completion Tokens	**5879**
思考链 Tokens	**5115**
max_tokens	**8000**
思考模式	**开启**