Formulaire Greedy + Autres

========== Kruskal: ==========

```python

def in_tree(v: int, tree: set[tuple[int, int]]) -> bool:
    for edge in tree:
        if v in edge: return True
    return False

def find_tree(v: int, trees: list[set[tuple[int, int]]]) -> int: #set[tuple[int, int]]:
    for i, t in enumerate(trees):
        if in_tree(v, t):
            return i
    s = f"{v} should be in trees since it contains all edges"
    raise ValueError(s)

def merge_tree(t1_idx: int, t2_idx: int, trees: list[set[tuple[int, int]]]):
        t1, t2 = trees[t1_idx], trees[t2_idx]
        trees.remove(t1)
        trees[t2_idx] = t1.union(t2)


def kruskal(A: list[list[int]]) -> set[tuple[int, int]]:  # sourcery skip: identity-comprehension
    """A: adjacency matrix, Return: MST"""
    N = len(A)
    if not A or not A[0]: return set()
    if N == 1: return {(0, 0)}

    trees, S = [], []  # S: sorted list of edges (w.r.t. weight)
    trees: list[set[tuple[int, int]]] = []
    for i in range(N):
        # list of trees at start : list of trees of length 1 i.e. each edge in a singleton
        for j in range(N):
            if A[i][j] == 0: continue  # (no edge)
            trees.append({(i, j)})
            S.append([(i, j), A[i][j]])

    S.sort(key=lambda kv: kv[1], reverse=True)  # min should be last
    F: set[tuple[int, int]] = set()  # forest which will hold/be the MST
    while S and len(F) < N:
        e, cost = S.pop()
        v1, v2 = e
        t1_idx, t2_idx = find_tree(v1, trees), find_tree(v2, trees)
        if trees[t1_idx] != trees[t2_idx]:
            F.add(e)
            merge_tree(t1_idx, t2_idx, trees)
    return F
```

2sum: "2 pointers"

```python
# find indices i,j such that nums[i] + nums[j] = target
def twoSum(self, nums: List[int], target: int) -> List[int]:
    left, right = 0, len(nums) - 1
    while nums[left] + nums[right] != target:
        if nums[left] + nums[right] < target:
            left += 1
        else:
            right -= 1
    return [left + 1, right + 1]
```

========== DFS & BFS: ==========

```python
class Node:
    """Node of binary tree. i.e. has 2 children until max depth is atteined"""
    MAX_DEPTH = 4  # hard-coded for test purposes
    # not max_depth -1 bcs values starts at 2**0 (1 | 2, 3 | 4, 5,6,7) so we added +1 as well
    MAX_VAL = 2 ** (MAX_DEPTH) - 1
    def __init__(self, val: int) -> None: self.val = val
    def get_adjacents(self) -> List[Self]:
        """Return: nodes adjacent to `self`. (i.e. children for a tree)"""
        val_left, val_right, val_top = self.val * 2, self.val * 2 + 1, max(self.val // 2, 1)  # last make graph undirecte
        return [] if val_right > Node.MAX_VAL else [Node(val_top), Node(val_left), Node(val_right)]


def BFS(root: Node, goal: int) -> Optional[Node]:
    visited: Set[int] = set()
    queue: List[Node] = [root]  # emulate queue behavior with list
    while len(queue) > 0:
        crt = queue.pop(0)  # queue => First in, First Out
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            # avoid cycle
            if node.val not in visited: queue.append(node)
            # adds to end => prioritize node on same depth
    return None

def DFS2(root: Node, goal: int) -> Optional[Node]:
    visited: Set[int] = set()
    stack: List[Node] = [root]
    while len(stack) > 0:
        crt = stack.pop(-1)
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            if node.val not in visited: stack.append(node)
    return None

def graph_traversal(root: Node, goal: int, is_bfs: bool):
    """if `is_bfs` search for `goal` with BFS else do it with DFS """
    visited: Set[int] = set()
    live_nodes: List[Node] = [root]
    def queue(): return live_nodes.pop(0)  # First In First Out (BFS)
    def stack(): return live_nodes.pop(-1) # First In Last Out (DFS)
    next_node = queue if is_bfs else stack
    while len(live_nodes) > 0:
        crt = next_node()
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            if node.val not in visited: live_nodes.append(node)
    return None
def DFS(root: Node, goal: int) -> Optional[Node]:
    out: Optional[Node] = None
    visited: Set[int] = set()
    def rec(crt: Node):
        nonlocal out
        visited.add(crt.val)
        for node in crt.get_adjacents():
            if node.val == goal:
                out = node
                return
            if node.val not in visited: rec(node)
            if out is not None: return
    rec(root)
    return out
if __name__ == "__main__":
    root = Node(1)
    found = graph_traversal(root=root, goal=16, is_bfs=True)
    print("Not Found" if found is None else f"Found: {found.val}")
```

d i.e ⟷

```python
def dijkstra(V: list[int], M: list[list[int | float]], start: int, goal: int):
    """start, goal: indices of start and goal vertex in V"""
    N = len(V)
    C, visited = [inf] * N, set()
    C[start] = 0
    path: list[int] = []
    pq: list[tuple[int, int]] = [(start, 0)] # node, cost

    def add_priority(node: int, cost: int):
        pq.append((node, cost))
        for i in range(len(pq) - 1, 0, -1):
            crt, next = pq[i], pq[i - 1]
            if crt[1] > next[1]: pq[i], pq[i - 1] = next, crt
            else: break

    def get_adjacent(node: int) -> list[int]:
        return [i for i in range(N) if i != node and isfinite(M[node][i])]

    enode = None
    while len(pq) > 0:
        enode = pq.pop()[0]
        if enode in visited: continue
        visited.add(enode)
        if enode == goal: return C, path
        for node in get_adjacent(enode):
            if node in visited: continue
            cost = M[enode][node] + C[enode]
            if cost < C[node]:
                C[node] = cost
                add_priority(node, cost) # type: ignore
    return C, path
```

================ Dijkstra ================

```python
def dijkstra(V: list[int], M: list[list[int | float]], start: int, goal: int):
    """start, goal: indices of start and goal vertex in V"""
    N = len(V)
    C, visited = [inf] * N, set()
    C[start] = 0
    path: list[int] = []
    pq: list[tuple[int, int]] = [(start, 0)] # node, cost

    def add_priority(node: int, cost: int):
        pq.append((node, cost))
        for i in range(len(pq) - 1, 0, -1):
            crt, next = pq[i], pq[i - 1]
            if crt[1] > next[1]: pq[i], pq[i - 1] = next, crt
            else: break

    def get_adjacent(node: int) -> list[int]:
        return [i for i in range(N) if i != node and isfinite(M[node][i])]

    enode = None
    while len(pq) > 0:
        enode = pq.pop()[0]
        if enode in visited: continue
        visited.add(enode)
        if enode == goal: return C, path
        for node in get_adjacent(enode):
            if node in visited: continue
            cost = M[enode][node] + C[enode]
            if cost < C[node]:
                C[node] = cost
                add_priority(node, cost) # type: ignore
    return C, path
```