VACUUM CLEANER

This project implements a **vacuum cleaner agent** on a 3x3 grid using Depth-First Search (DFS) to find a sequence of actions that cleans all dirty cells.

Environment and state

The world is a 3x3 grid represented by `state[3][3]` where:

- `0`: empty cell
- `1`: dirt
- `2`: vacuum cleaner
- `3`: vacuum cleaner and dirt in the same cell

A `Node` stores:

- `state`: the 3x3 grid configuration
- `num_dirt`: remaining dirty cells
- `vc_pos`: current position of the vacuum cleaner (`x`, `y` in 0..2)
- `parent`: pointer to the parent node
- `op_parent`: action used in the parent to generate this node

Two nodes are considered equal if their `state` matrices are the same.

Actions (operators)

The agent can perform five actions:

- `SUCK`: clean dirt in the current cell
- `LEFT`, `RIGHT`: move horizontally within the grid bounds
- `UP`, `DOWN`: move vertically within the grid bounds

The function `posible(Op oper)` checks:

- Movement actions: ensure the new position remains inside the 3x3 grid.
- `SUCK`: only allowed if the current cell contains dirt (coded as `3`).

Each action updates:

- The grid (`state`) to move the robot and adjust dirt/empty codes.
- The robot position `vc_pos`.
- `num_dirt` (decremented when sucking).
- `op_parent` and `parent` so the search tree can be reconstructed.

Input and initial state

In `main()` the program reads the initial grid from standard input:

- `'_'` → empty (`0`)
- `'*'` → dirt (`1`), increments `num_dirt`
- `'R'` → robot (`2`), sets its initial coordinates

With this, it builds `initial_state` and starts a `DFS` search object, then calls `dfs_search()`.

DFS search process

The `DFS` struct maintains:

- `open`: list of nodes yet to be explored
- `closed`: list of already expanded nodes
- `actual`: current node being expanded

`dfs_search()` loop:

1. While `open` is not empty and no solution is found:
    - Take the first node from `open` as `actual`.
    - If `actual.num_dirt == 0`, this is a goal state (no dirt left):
        - Call `recover_solution(actual)` to reconstruct and print the action sequence.
    - Otherwise, call `generate_nodes()` to expand this node.

If `open` becomes empty without finding a node with `num_dirt == 0`, the program prints that no solution was found.

Node expansion and duplicate detection

`generate_nodes()`:

- Adds `actual` to `closed`.
- For each possible operator (`SUCK`, `LEFT`, `RIGHT`, `UP`, `DOWN`):
    - Checks if it is applicable with `posible(oper)`.
    - Copies `actual` into a new node `aux`, applies the action (updates grid, robot position, dirt count, parent, and operator).
    - Uses `in_a_list(aux)` to check if a node with the same `state` is already in `open` or `closed`.
    - If not found, pushes `aux` into `open`.

`in_a_list` uses the `operator==` of `Node` to compare states, which prevents exploring the same grid configuration multiple times and avoids simple cycles.

Recovering and printing the solution

When a goal state is reached, `recover_solution(Node solution)`:

- Follows the `parent` pointers backwards from the solution node to the root, storing each `op_parent` in a list (front-insert to get correct order).

- Then iterates over this list and calls `decodify(oper)` to print the actions as text lines:
    - `SUCK`, `LEFT`, `RIGHT`, `UP`, `DOWN`

This produces a readable plan describing step by step how the robot moves and sucks to clean the entire 3x3 grid.

Overall idea

In short, the program models the classic AI vacuum cleaner world on a 3x3 grid and uses DFS over the state space:

- Reads an initial configuration with robot and dirt.
- Systematically explores possible moves and suck actions.
- Avoids revisiting equivalent states.
- Prints a sequence of actions that leads to a completely clean grid, if such a sequence exists.