LISV Laboratoire d'ingénierie des systèmes de Versailles
UVSQ université PARIS-SACLAY

IA Planning TP 1

Halim Djerroud

revision 1.0

TP 1: Introduction to Automated Planning with the 8-puzzle

Objectives

- Discover the concept of automated planning.
- Manipulate basic concepts: state, action, goal, plan.
- Understand the difference between BFS and DFS.

Problem Presentation: The 8-puzzle

A 3×3 grid with 8 numbered tiles and one empty cell (noted as b). An **action** moves an adjacent tile into the empty cell. A **plan** is a sequence of actions leading from the initial state to the goal state.

Exercise 1: Define the States

1. Represent a puzzle state as a 3×3 array.
2. Write an initial state and a simple goal state (for example, with two tiles swapped).
3. How many different states are possible? How many are reachable?

Solution: A state is represented by a 3×3 matrix, for example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & b \end{bmatrix}$$

Possible initial state:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & b & 8 \end{bmatrix}$$

Goal state:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & b \end{bmatrix}$$

Total number of permutations: $9! = 362\,880$. But only half are reachable $\Rightarrow 181\,440$ valid states.

Solution: Why are only half of the configurations reachable?

1) Linearization and inversions. We "unroll" the grid by reading the cells row by row (reading order) and *ignore* the empty cell b. We obtain a permutation of tiles $1, \ldots, 8$. We call an *inversion* any pair (i, j) with $i < j$ but the tile at position i is numerically *greater* than the tile at position j. Let $I(s)$ denote the number of inversions in state s.

2) Invariance of inversion parity for a 3×3 grid (odd width). A legal move exchanges b with a neighboring tile (up, down, left, right).

- *Horizontal move (left/right).* In the "without b" list, the relative order of tiles doesn't change, since b is not counted. Therefore $I(s)$ doesn't change.
- *Vertical move (up/down).* When linearizing row by row, two vertically adjacent cells are separated by 3 positions. When a tile moves up/down one row, in the "without b" list it *jumps exactly two tiles.* The number of inversions therefore varies by ± 2 (even number).

Conclusion: at each move, $I(s)$ varies by 0 or 2 $\Rightarrow$ **the parity of $I(s)$ is invariant** (preserved).

3) Consequence for solvability. The standard goal state $(1, 2, 3 \,/\, 4, 5, 6 \,/\, 7, 8, b)$ has $I = 0$ (even). Since parity is preserved, a state is **solvable** iff its permutation (without b) has an **even number of inversions**. States with odd parity are **unreachable** from the goal (and vice versa).

4) Counting. There are 9 possible positions for b and 8! possible arrangements of the tiles. By permutation theory, **exactly half** of the permutations of 8 elements are even, the other half odd. Therefore, for each position of b, $\frac{8!}{2}$ states are solvable. In total:

$$\#\text{solvable states} \;=\; 9 \times \frac{8!}{2} \;=\; \frac{9!}{2} \;=\; 181\,440.$$

5) Example of unsolvable state. Swapping 1 and 2 in the goal state gives a permutation with a single inversion (odd) $\Rightarrow$ *unsolvable.*

Exercise 2: Define the Actions

1. Describe in your own words an action "move a tile".
2. Give all possible actions from a state with b at the center, at an edge, and in a corner.
3. Explain the adjacency constraint.

Solution: An action consists of exchanging the empty cell b with a neighboring tile (up, down, left, right).

- b at center: 4 possible actions.
- b on an edge (non-corner): 3 possible actions.
- b in a corner: 2 possible actions.

Adjacency is orthogonal (no diagonal moves).

Exercise 3: Search for a Plan by Hand (advanced version)

Initial state:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & 7 & b \end{bmatrix}$$

Goal:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & b \end{bmatrix}$$

1. Find a short plan (sequence of actions) to solve this problem.
2. Verify each intermediate state.
3. Are there multiple possible plans?
4. Compare what BFS and DFS would produce.

Solution: Minimal plan (4 moves):

1. Move 7 to the right:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & b & 7 \end{bmatrix}$$
2. Move 8 to the right:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ b & 8 & 7 \end{bmatrix}$$
3. Move 7 down:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & b \end{bmatrix}$$
4. Wait, let me correct this. Move 7 up:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & b & 8 \end{bmatrix}$$
Then move 8 left:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & b \end{bmatrix}$$

Action sequence: `move(7), move(8), move(7), move(8)`.

State-by-state verification: Each move respects the adjacency constraint (empty cell exchanged with a neighboring tile).

Existence of other plans: Yes, some plans also lead to the goal but in 5 or 6 moves. Only the plan above is minimal (4 moves).

Connection to BFS/DFS:

- BFS will explore states level by level and find this 4-move plan (optimal).
- DFS might find a longer solution (e.g., 6–7 moves) depending on exploration order, and may even get lost in useless branches.

Exercise 4: BFS vs DFS (concepts)

1. Explain the difference between BFS and DFS.
2. Which one guarantees the shortest plan?
3. Compare memory, time, and completeness.

Solution:

- **BFS** explores level by level, **DFS** dives into a branch.
- BFS guarantees the shortest plan (uniform cost), DFS does not.
- BFS is complete but memory-expensive; DFS uses little memory but can be incomplete and give a long plan.

Exercise 5: Heuristics (bonus)

1. Define the misplaced tiles heuristic.
2. Define the Manhattan distance.
3. Which is more informative?

Solution:

- Misplaced tiles = number of tiles that are not in their final position.
- Manhattan distance = sum of L_1 distances from each tile to its goal position.
- Manhattan distance is more informative as it gives a finer estimate.

Exercise 6: State Representation in Python

1. Represent an 8-puzzle state as a list of lists in Python.
2. Write a function `display(state)` that prints the 3×3 grid nicely.
3. Test this function with the initial state and goal state.

Exercise 7: Successor Generation

1. Write a function `successors(state)` that returns the list of states obtained by moving a tile into the empty cell.
2. Test the function on the initial state.

Exercise 8: BFS and DFS in Python

1. Write a function `bfs(initial_state, goal_state)` that returns a plan (sequence of states).
2. Write a function `dfs(initial_state, goal_state, max_depth)` that returns a plan if found.
3. Compare the results on a small example.

Solution exo 6, 7 and 8 :

```
from collections import deque
import copy

def display(state):
    for row in state:
        print(*row)
    print()

initial_state = [
    [1, 2, 3],
    [4, 5, 6],
    [0, 7, 8]
]

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

def state_to_str(state):
    return "".join(str(x) for row in state for x in row)

def find_empty_cell(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
```

```
                return i, j

def successors(state):
    i, j = find_empty_cell(state)
    moves = [(-1,0),(1,0),(0,-1),(0,1)]
    succ = []
    for di, dj in moves:
        ni, nj = i + di, j + dj
        if 0 <= ni < 3 and 0 <= nj < 3:
            new_state = copy.deepcopy(state)
            new_state[i][j], new_state[ni][nj] = new_state[ni][nj], new_state[i][j]
            succ.append(new_state)
    return succ

def bfs(start, goal):
    start_s, goal_s = state_to_str(start), state_to_str(goal)
    queue = deque([(start, [])])
    visited = {start_s}
    while queue:
        state, plan = queue.popleft()
        if state_to_str(state) == goal_s:
            return plan + [state]
        for s in successors(state):
            st = state_to_str(s)
            if st not in visited:
                visited.add(st)
                queue.append((s, plan + [state]))
    return None


def dfs(start, goal):
    start_s, goal_s = state_to_str(start), state_to_str(goal)
    stack = [(start, [])]
    visited = {start_s}
    while stack:
        state, plan = stack.pop()
        if state_to_str(state) == goal_s:
            return plan + [state]
        for s in successors(state):
            st = state_to_str(s)
            if st not in visited:
                visited.add(st)
                stack.append((s, plan + [state]))
    return None


solution_bfs = bfs(initial_state, goal_state)
if solution_bfs :
    print("BFS Solution find in", len(solution)-1, "mouvements :")
    for step in solution:
        display(step)
else:
    print("no solution")

solution_dfs = dfs(initial_state, goal_state)
if solution_dfs :
    print("DFS Solution find in", len(solution)-1, "mouvements :")
    for step in solution:
        display(step)
```

```
    else:
        print("no solution")
```

Exercise 9: Heuristics (bonus)

1. Write a function `h_misplaced(state, goal)` = number of misplaced tiles.
2. Write a function `h_manhattan(state, goal)` = sum of Manhattan distances.
3. Test both heuristics on the initial state.

Exercise 10: Implement A*

1. Implement the A* algorithm in Python with a priority queue (`heapq`).
2. Use as heuristic: (a) misplaced tiles, (b) Manhattan distance.
3. Compare the length of plans found with BFS.

Solution (simplified example):

```
import heapq

def a_star(start, goal, h):
    start_t, goal_t = state_to_tuple(start), state_to_tuple(goal)
    open_list = [(h(start, goal), 0, start, [])]
    visited = {start_t: 0}
    while open_list:
        f, g, state, plan = heapq.heappop(open_list)
        if state_to_tuple(state) == goal_t:
            return plan + [state]
        for s in successors(state):
            st = state_to_tuple(s)
            g2 = g + 1
            if st not in visited or g2 < visited[st]:
                visited[st] = g2
                heapq.heappush(open_list,
                    (g2 + h(s, goal), g2, s, plan + [state]))
    return None
```

Exercise 11: Experimental Comparison

1. Measure execution time and number of nodes explored for BFS, DFS, and A*.
2. Create a comparative table for different initial states.
3. Conclude on the advantages of each algorithm.

Solution (idea):

- BFS explores many states but always finds the optimal plan.
- DFS explores little memory but risks missing a solution.
- A* quickly finds an optimal solution if the heuristic is well chosen.

Exercise 12: Plan Visualization

1. Write a function `display_plan(plan)` that displays each state in the found plan, with a delay (e.g., 0.5s).
2. Test with a plan found by BFS.

Solution (example):

```
import time

def display_plan(plan):
    for state in plan:
        display(state)
        time.sleep(0.5)
```

Exercise 13: Extension to the 15-puzzle

1. Adapt the functions for a 4×4 grid with 15 tiles and 1 empty cell.
2. Test BFS (on small shuffles) and A* (with Manhattan heuristic).

Solution (idea):

- Just change the grid size (4 instead of 3).
- BFS quickly becomes impractical due to combinatorial explosion.
- A* remains usable with a good heuristic.

Exercise 14: Random State Generation

1. Write a function that generates a random initial state by applying n moves from the goal state.
2. Use this function to automatically test your algorithms.

Solution (example):

```
import random

def shuffle(state, n=20):
    current = copy.deepcopy(state)
    for _ in range(n):
        succ = successors(current)
        current = random.choice(succ)
    return current
```