Reservoir Distribution with Valves

Assignment 1 - CSL304 AI

1 Problem Overview

Problem Statement

Objective: Determine the sequence of valve operations to transform initial water distribution into target distribution while minimizing the number of valve openings.
Key Constraints:

- 3 reservoirs connected by bidirectional valves
- Water flows until destination is full OR source reaches 20% safety threshold
- Only one valve can be open at a time
- Total water is conserved
- Each reservoir must maintain 20% of its capacity

1.1 Problem Formulation

Given:

- Capacities: (C_1, C_2, C_3)
- Initial amounts: (I_1, I_2, I_3)
- Target amounts: (T_1, T_2, T_3)
- Safety constraint: Each reservoir must maintain $0.2 \times C_i$

State Representation:

$$\text{State} = (w_1, w_2, w_3) \text{ where } w_i \text{ is current water in reservoir } i$$

Water Transfer Formula:

$$\text{Transfer}_{i \to j} = \min(\max(0, w_i - 0.2 \times C_i), \max(0, C_j - w_j))$$

2 Solution Methodology

Solution Approach

Three Search Algorithms Implemented:

1. **BFS:** Guarantees minimum number of valve operations (optimal solution)
2. **DFS:** Explores depth-first with backtracking (may find different valid path)
3. **A*:** Uses heuristic to guide search towards goal state efficiently

3 Algorithm Implementation

3.1 State Space Representation

Algorithm Details

State Management
Each state is represented as a tuple of water amounts rounded to 1 decimal place for precision handling.

Algorithm 1 State Rounding and Action Application

function round_state(state):
return tuple(round(x + 1e-9, 1) for x in state)
function apply_action(state, caps, i, j):
src_available ← max(0, state[i] - 0.2 × caps[i])
dst_capacity ← max(0, caps[j] - state[j])
transfer ← min(src_available, dst_capacity)
if transfer > 1e-9 **then**
new_state[i] ← state[i] - transfer
new_state[j] ← state[j] + transfer
return new_state, transfer
else
return None, 0
end if

3.2 BFS Implementation

```
def bfs(start, target, caps):
    start = round_state(start)
    target = round_state(target)
    q = deque([start])
    parents = {start: (None, None)}

    while q:
        s = q.popleft()
        if is_goal(s, target):
            path = reconstruct(parents, s)
```

```
            return path, s

        for (i,j), new_s, transfer in actions_from_state(s, caps):
            if new_s not in parents:
                parents[new_s] = (s, (i,j))
                q.append(new_s)

    return None, None
```

Listing 1: Breadth-First Search Implementation

BFS Characteristics:

- **Optimal:** Guarantees minimum number of valve operations
- **Complete:** Will find solution if one exists
- **Memory intensive:** Stores all nodes at current depth

3.3 DFS Implementation

```
def dfs(start, target, caps, max_nodes=20000):
    start = round_state(start)
    target = round_state(target)
    stack = [start]
    parents = {start: (None, None)}
    visited = set([start])
    nodes = 0

    while stack and nodes < max_nodes:
        s = stack.pop()
        nodes += 1
        if is_goal(s, target):
            return reconstruct(parents, s), s

        for (i,j), new_s, transfer in actions_from_state(s, caps):
            if new_s not in visited:
                visited.add(new_s)
                parents[new_s] = (s, (i,j))
                stack.append(new_s)

    return None, None
```

Listing 2: Depth-First Search Implementation

DFS Characteristics:

- **Memory efficient:** Uses less memory than BFS
- **Not optimal:** May find longer path to solution
- **Fast for deep solutions:** Good when solution is far from start

3.4 A* Implementation with Heuristic

Algorithm Details

A* Heuristic Design

The heuristic function estimates the minimum number of valve operations needed:

$$h(\text{state}) = \left\lceil \frac{\sum_{i=1}^{3} |w_i - t_i|/2}{\max(\text{possible transfer per operation})} \right\rceil$$

Where possible transfer per operation $\approx 0.8 \times \max(C_i)$

```python
def astar(start, target, caps):
    start = round_state(start)
    target = round_state(target)

    def heuristic(state):
        # Total "distance" of water between current and target
        diff = sum(abs(si - ti) for si, ti in zip(state, target))
        needed_transfer = diff / 2.0
        max_transfer = 0.8 * max(caps)
        return math.ceil(needed_transfer / max_transfer)

    open_heap = []
    gscore = {start: 0}
    parents = {start: (None, None)}
    fscore = {start: heuristic(start)}
    heapq.heappush(open_heap, (fscore[start], start))
    closed = set()

    while open_heap:
        _, s = heapq.heappop(open_heap)
        if s in closed:
            continue
        if is_goal(s, target):
            return reconstruct(parents, s), s
        closed.add(s)

        for (i,j), new_s, transfer in actions_from_state(s, caps):
            tentative_g = gscore[s] + 1  # each action costs 1
            if new_s in closed:
                continue
            if new_s not in gscore or tentative_g < gscore[new_s]:
                gscore[new_s] = tentative_g
                parents[new_s] = (s, (i,j))
                f = tentative_g + heuristic(new_s)
                heapq.heappush(open_heap, (f, new_s))

    return None, None
```

Listing 3: A* Search with Heuristic

A* Heuristic Properties:

- **Admissible:** Never overestimates the actual cost
- **Consistent:** $h(n) \leq c(n, n') + h(n')$ for all neighbors
- **Informative:** Provides good guidance towards goal

4 Sample Execution & Results

Results & Analysis

Input Example:

- Capacities: (8.0, 5.0, 3.0)
- Initial: (8.0, 0.0, 0.0)
- Target: (2.4, 5.0, 0.6)

4.1 Algorithm Comparison

Algorithm	Path Found	Operations	Optimality
BFS	1→2, 1→3, 3→1	3	Optimal
DFS	1→3, 3→2, 1→2	3	Optimal (lucky)
A*	1→2, 1→3, 3→1	3	Optimal

4.2 Step-by-Step Execution (BFS Solution)

Initial State: (8.0, 0.0, 0.0)

1. **Open valve (1→2):**
   - Available from reservoir 1: $8.0 - 0.2 \times 8.0 = 6.4$
   - Capacity in reservoir 2: $5.0 - 0.0 = 5.0$
   - Transfer: $\min(6.4, 5.0) = 5.0$
   - New state: (3.0, 5.0, 0.0)
2. **Open valve (1→3):**
   - Available from reservoir 1: $3.0 - 0.2 \times 8.0 = 1.4$
   - Capacity in reservoir 3: $3.0 - 0.0 = 3.0$
   - Transfer: $\min(1.4, 3.0) = 1.4$
   - New state: (1.6, 5.0, 1.4)
3. **Open valve (3→1):**
   - Available from reservoir 3: $1.4 - 0.2 \times 3.0 = 0.8$
   - Capacity in reservoir 1: $8.0 - 1.6 = 6.4$
   - Transfer: $\min(0.8, 6.4) = 0.8$
   - Final state: (2.4, 5.0, 0.6)

5 Key Implementation Features

5.1 Helper Functions

```python
def is_goal(state, target, tol=1e-6):
    return all(abs(state[i] - target[i]) < tol for i in range(3))

def actions_from_state(state, caps):
    acts = []
    for i in range(3):
        for j in range(3):
            if i == j:
                continue
            new_state, transfer = apply_action(state, caps, i, j)
            if new_state is not None:
                acts.append(((i, j), new_state, transfer))
    return acts

def reconstruct(parents, end_state):
    path = []
    cur = end_state
    while parents[cur][0] is not None:
        act = parents[cur][1]  # (i,j)
        path.append(act)
        cur = parents[cur][0]
    path.reverse()
    return path
```

Listing 4: Core Helper Functions

6 Complexity Analysis

Algorithm Details

Time & Space Complexity
Let S be the number of possible states and b be the branching factor.

- **BFS:** Time $O(S)$, Space $O(S)$ - explores states level by level
- **DFS:** Time $O(S)$, Space $O(d)$ where d is maximum depth
- **A*:** Time $O(S \log S)$, Space $O(S)$ - priority queue operations

State Space Size: Each reservoir can have water amounts in increments of 0.1, so state space is bounded but can be large for high-precision requirements.

7 Algorithm Insights

1. **Precision Handling:** Floating-point arithmetic requires careful rounding to avoid precision errors in state representation.

2. **Safety Constraints:** The 20% minimum requirement significantly constrains the action space and ensures reservoir safety.

3. **Heuristic Quality:** A* performs well because the heuristic accurately estimates the minimum number of operations based on water redistribution needs.

4. **Optimality Trade-offs:** BFS guarantees optimality but uses more memory, while DFS is memory-efficient but may find suboptimal solutions.

8 Conclusion

This solution demonstrates the application of three fundamental search algorithms to a constrained optimization problem. The reservoir distribution problem showcases how different search strategies can yield optimal or near-optimal solutions with varying computational trade-offs.

Key Takeaways:

- BFS guarantees optimal solutions for unweighted problems
- A* with well-designed heuristics combines optimality with efficiency
- DFS provides memory-efficient exploration but may sacrifice optimality
- Proper state representation and precision handling are crucial for floating-point domains