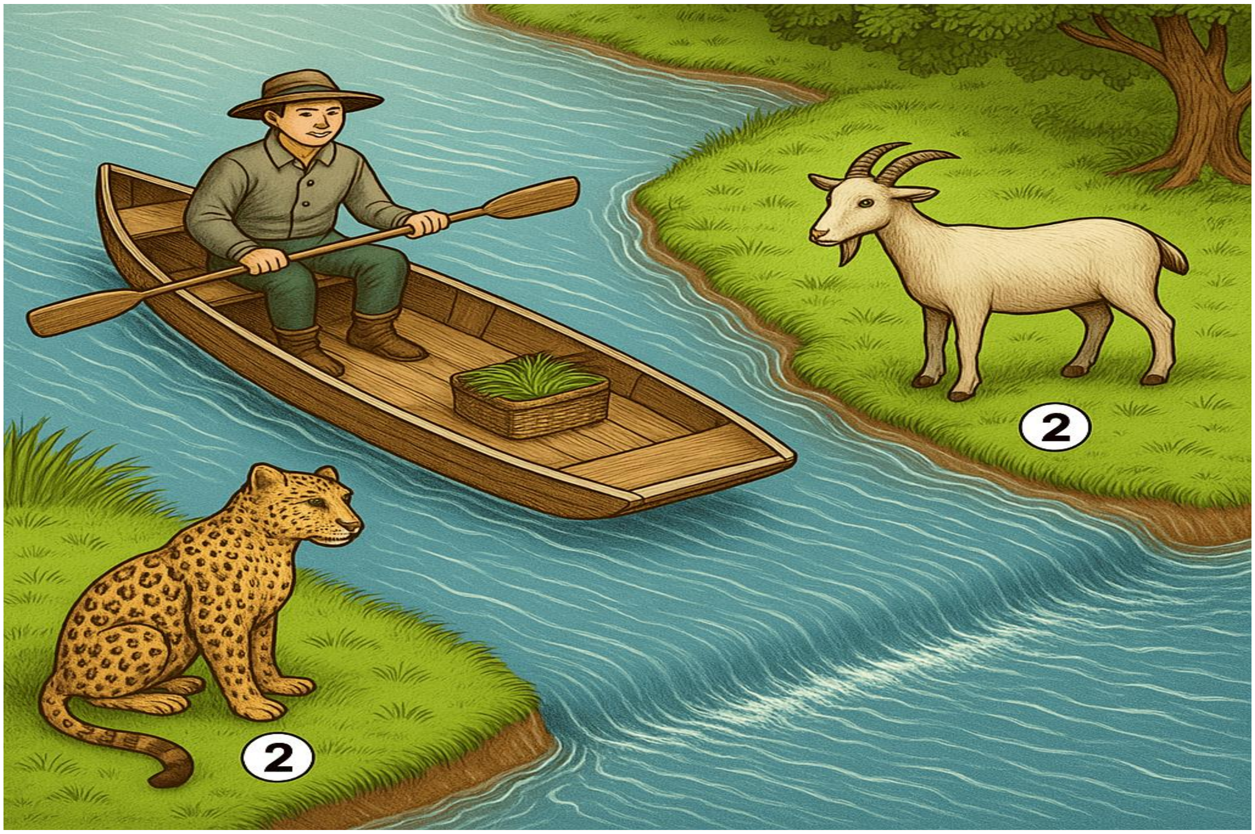
Solving the River-Crossing Puzzle using BFS and DFS



1. Understanding the Problem

The River-Crossing Puzzle

A man needs to transport three items—a **leopard, a goat, and a bundle of grass**—across a river using a small boat. However, he can only carry **one item at a time** and must ensure that no conflicts arise when he is not present.

Constraints (Dangerous Situations)

- The **leopard will eat the goat** if left together without the man.
- The **goat will eat the grass** if left together without the man.
- The man must always be present to prevent conflicts.
- The boat can only carry the **man and one item at a time**.

Objective

- **Initial State:** Everything starts on the **left** riverbank.
- **Goal State:** Everything must safely reach the **right** riverbank.

Key Questions to Solve the Puzzle

1. In what order should the man transport the items?
2. How can we avoid unsafe states where something gets eaten?
3. Can we represent this problem in a **graph structure** for an algorithmic solution?

2. Representing the Problem as a State Space

Using Binary Encoding

Each state is a **4-bit tuple (M, L, G, Gr)**, where:

- **0 = Left bank**, **1 = Right bank**

Bit Position	Represents
1st (M)	Man
2nd (L)	Leopard
3rd (G)	Goat
4th (Gr)	Grass

Key States

- **Initial State** → (0,0,0,0) → Everything starts on the left bank.
- **Goal State** → (1,1,1,1) → Everything is safely moved to the right bank.

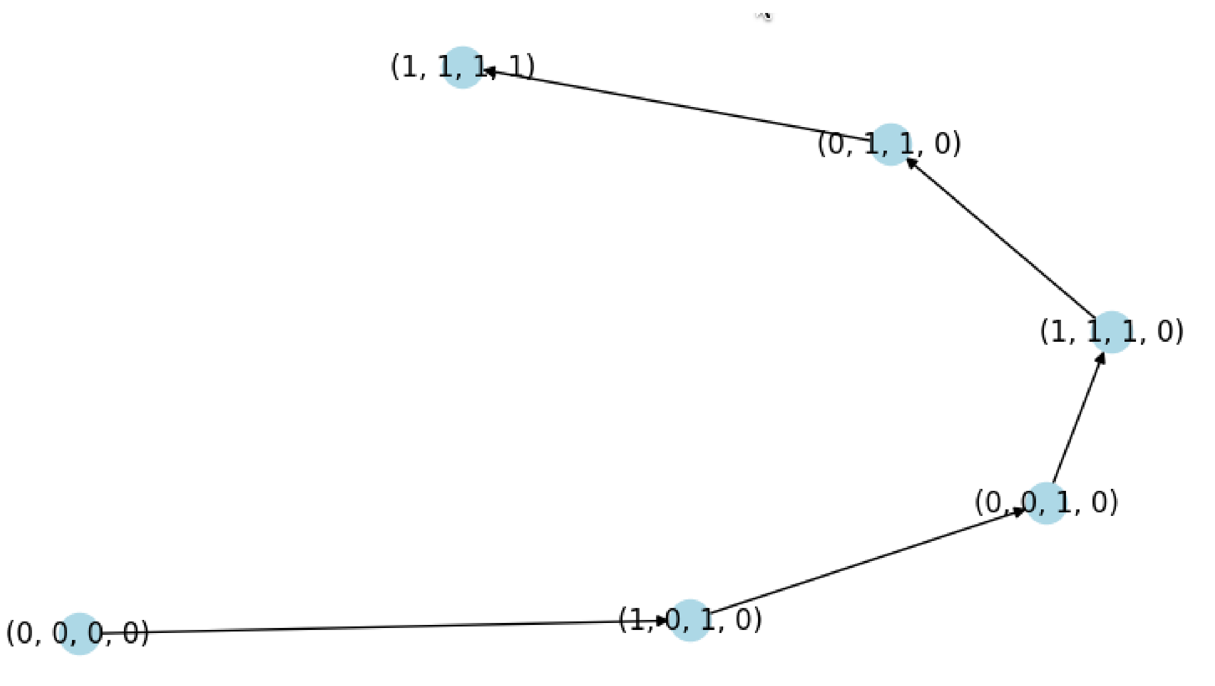
Example State Transitions

State	Meaning
(0,0,0,0)	Everything on the left (Start)
(1,0,1,0)	Man & Goat crossed to the right
(0,0,1,0)	Man returned alone
(1,1,1,0)	Man took Leopard across
(0,1,1,0)	Man returned alone
(1,1,1,1)	Man took Grass across (Goal)

Graph Representation

To solve this as a **graph traversal problem**, we create a **state transition graph** where:

- **Nodes** represent **valid states**.
- **Edges** represent **valid moves** (where the man crosses with or without an item).
- The goal is to find a path from **(0,0,0,0)** to **(1,1,1,1)**.



Graph Explanation

- Each **node** is a state (M, L, G, Gr).
- Each **edge** represents a valid move.
- The path from (0,0,0,0) to (1,1,1,1) shows a **possible solution sequence**.

3. Defining Valid Moves

In the river-crossing puzzle, the valid moves must adhere to the constraints to prevent unsafe states. The rules are:

1. The **man** is the only one who can operate the boat.

2. The **leopard cannot be left alone with the goat** (the leopard will eat the goat).
3. The **goat cannot be left alone with the grass** (the goat will eat the grass).
4. Only the **man and one other entity** (leopard, goat, or grass) can cross at a time.
5. The **man can also cross alone** if needed.

List of Valid Moves

From any given state, the following moves are valid if they do not result in an unsafe state:

- **Move Alone** – The man crosses the river by himself.
- **Take Leopard** – The man takes the leopard across.
- **Take Goat** – The man takes the goat across.
- **Take Grass** – The man takes the grass across.

State Transition Examples

Each move results in a new state that must be checked for validity:

- **(M, L, G, G)** → **(M, _, G, G)** → Invalid (leopard will eat goat)
- **(M, L, G, G)** → **(M, L, G, _)** → Valid (grass is left alone)

4. Implementing the Search Algorithms

To solve the river-crossing puzzle, we implement **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

- **BFS** explores all possible moves level by level, ensuring the shortest path is found.
- **DFS** explores one path deeply before backtracking, which may not always yield the shortest path.

Implementing the BFS Algorithm

Concept

- **Explores all possible moves** at the current level before moving deeper.
- Uses a **queue (FIFO - First In, First Out)** to track the states.
- **Guarantees finding the shortest solution** if one exists.

BFS Implementation

```
from collections import deque
from tabulate import tabulate

# State Representation: (Man, Leopard, Goat, Grass)
# 0 = Left Bank, 1 = Right Bank
initial_state = (0, 0, 0, 0) # All on the left bank
goal_state = (1, 1, 1, 1) # All on the right bank

def is_valid_state(state):
"""Ensures the leopard is never left alone with the goat, and the goat is never left alone with the grass."""
man, leopard, goat, grass = state
# Goat must not be alone with leopard, and grass must not be alone with goat
if (leopard == goat and goat != man) or (goat == grass and grass != man):
return False
return True

def get_neighbors(state):
"""Generates valid next states from the current state."""
man, leopard, goat, grass = state
neighbors = []
# Possible moves: Man can travel alone or take one item
possible_moves = [(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)] # (Leopard, Goat, Grass)

for move in possible_moves:
leopard_move, goat_move, grass_move = move

# Move Man and the selected item(s) to the other side
new_man = 1 - man # Flip side
new_leopard = leopard + leopard_move if man == 0 else leopard - leopard_move
new_goat = goat + goat_move if man == 0 else goat - goat_move
new_grass = grass + grass_move if man == 0 else grass - grass_move

# Ensure all new positions are within valid bounds (0 or 1)
if not (0 <= new_leopard <= 1 and 0 <= new_goat <= 1 and 0 <= new_grass <= 1):
continue

new_state = (new_man, new_leopard, new_goat, new_grass)
if is_valid_state(new_state):
neighbors.append(new_state)

return neighbors

def bfs():
"""Performs BFS to find the shortest solution path."""
queue = deque([(initial_state, [])])
visited = set([initial_state])

while queue:
state, path = queue.popleft()
```

```
if state == goal_state:
return path + [state]

for neighbor in get_neighbors(state):
if neighbor not in visited:
visited.add(neighbor)
queue.append((neighbor, path + [state]))

return None # No solution found

def get_move_description(from_state, to_state, move):
"""Generates a readable description of a move."""
man_from, leopard_from, goat_from, grass_from = from_state
man_to, leopard_to, goat_to, grass_to = to_state
man_move, leopard_move, goat_move, grass_move = move

actions = []
if leopard_move:
actions.append("Leopard")
if goat_move:
actions.append("Goat")
if grass_move:
actions.append("Grass")

if not actions:
return "Man crosses alone."

action_str = ', '.join(actions)
direction = "right" if man_from == 0 else "left"
return f"Man takes {action_str} to the {direction} bank."

def print_solution(solution):
"""Formats and prints the solution using a table."""
if solution:
headers = ["Step", "Man", "Leopard", "Goat", "Grass", "Action"]
table = []

# Add the initial state with "Initial State" description
table.append(["", solution[0][0], solution[0][1], solution[0][2], solution[0][3], "Initial State"])

for i in range(len(solution) - 1):
prev_state = solution[i]
next_state = solution[i + 1]
move = tuple(next_state[j] - prev_state[j] for j in range(4))
action_description = get_move_description(prev_state, next_state, move)
table.append([i + 1, next_state[0], next_state[1], next_state[2], next_state[3], action_description])

# Add final goal state description
table.append(["", solution[-1][0], solution[-1][1], solution[-1][2], solution[-1][3], "Reached goal state"])
```

```
print("\nSolution found:")
print(tabulate(table, headers=headers, tablefmt="grid"))
else:
print("No solution found.")

# Run BFS to solve the puzzle
solution = bfs()

# Display the solution in a human-readable format
print_solution(solution)
```

Output

```
Solution found:
+--------+-------+-----------+--------+---------+-------------------------------------+
| Step   |   Man |   Leopard |   Goat |   Grass | Action                              |
+========+=======+===========+========+=========+=====================================+
|        |     0 |         0 |      0 |       0 | Initial State                       |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 1      |     1 |         0 |      1 |       0 | Man takes Goat to the right bank.   |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 2      |     0 |         0 |      1 |       0 | Man crosses alone.                  |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 3      |     1 |         1 |      1 |       0 | Man takes Leopard to the right bank. |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 4      |     0 |         1 |      0 |       0 | Man takes Goat to the left bank.    |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 5      |     1 |         1 |      0 |       1 | Man takes Grass to the right bank.  |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 6      |     0 |         1 |      0 |       1 | Man crosses alone.                  |
+--------+-------+-----------+--------+---------+-------------------------------------+
| 7      |     1 |         1 |      1 |       1 | Man takes Goat to the right bank.   |
+--------+-------+-----------+--------+---------+-------------------------------------+
|        |     1 |         1 |      1 |       1 | Reached goal state                  |
+--------+-------+-----------+--------+---------+-------------------------------------+
```

Implementing the DFS Algorithm
Concept

- **Explores one path deeply** before backtracking.
- Uses a **stack (LIFO - Last In, First Out)**.
- Can **get stuck in loops** unless properly managed.

DFS Implementation

```
from tabulate import tabulate

# State Representation: (Man, Leopard, Goat, Grass)
```

```
# 0 = Left Bank, 1 = Right Bank
initial_state = (0, 0, 0, 0) # All on the left bank
goal_state = (1, 1, 1, 1) # All on the right bank

def is_valid_state(state):
"""Ensures the leopard is never left alone with the goat, and the goat is never left alone with the grass."""
man, leopard, goat, grass = state
# Goat must not be alone with leopard, and grass must not be alone with goat
if (leopard == goat and goat != man) or (goat == grass and grass != man):
return False
return True

def get_neighbors(state):
"""Generates valid next states from the current state."""
man, leopard, goat, grass = state
neighbors = []
# Possible moves: Man can travel alone or take one item
possible_moves = [(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)] # (Leopard, Goat, Grass)

for move in possible_moves:
leopard_move, goat_move, grass_move = move

# Move Man and the selected item(s) to the other side
new_man = 1 - man # Flip side
new_leopard = leopard + leopard_move if man == 0 else leopard - leopard_move
new_goat = goat + goat_move if man == 0 else goat - goat_move
new_grass = grass + grass_move if man == 0 else grass - grass_move

# Ensure all new positions are within valid bounds (0 or 1)
if not (0 <= new_leopard <= 1 and 0 <= new_goat <= 1 and 0 <= new_grass <= 1):
continue

new_state = (new_man, new_leopard, new_goat, new_grass)
if is_valid_state(new_state):
neighbors.append(new_state)

return neighbors

def dfs():
"""Performs DFS to find a solution path."""
stack = [(initial_state, [])] # Stack for DFS
visited = set()

while stack:
state, path = stack.pop()

if state == goal_state:
return path + [state]

if state not in visited:
visited.add(state)
for neighbor in get_neighbors(state):
```

```
stack.append((neighbor, path + [state]))

return None # No solution found

def get_move_description(from_state, to_state, move):
"""Generates a readable description of a move."""
man_from, leopard_from, goat_from, grass_from = from_state
man_to, leopard_to, goat_to, grass_to = to_state
man_move, leopard_move, goat_move, grass_move = move

actions = []
if leopard_move:
actions.append("Leopard")
if goat_move:
actions.append("Goat")
if grass_move:
actions.append("Grass")

if not actions:
return "Man crosses alone."

action_str = ', '.join(actions)
direction = "right" if man_from == 0 else "left"
return f"Man takes {action_str} to the {direction} bank."

def print_solution(solution):
"""Formats and prints the solution using a table."""
if solution:
headers = ["Step", "Man", "Leopard", "Goat", "Grass", "Action"]
table = []

# Add the initial state with "Initial State" description
table.append(["", solution[0][0], solution[0][1], solution[0][2], solution[0][3], "Initial State"])

for i in range(len(solution) - 1):
prev_state = solution[i]
next_state = solution[i + 1]
move = tuple(next_state[j] - prev_state[j] for j in range(4))
action_description = get_move_description(prev_state, next_state, move)
table.append([i + 1, next_state[0], next_state[1], next_state[2], next_state[3], action_description])

# Add final goal state description
table.append(["", solution[-1][0], solution[-1][1], solution[-1][2], solution[-1][3], "Reached goal state"])

print("\nSolution found:")
print(tabulate(table, headers=headers, tablefmt="grid"))
else:
print("No solution found.")

# Run DFS to solve the puzzle
solution = dfs()
```

```
# Display the solution in a human-readable format
print_solution(solution)
```

Output

```
Solution found:
+--------+-------+----------+--------+---------+------------------------------------+
| Step   |   Man |  Leopard |   Goat |   Grass | Action                             |
+========+=======+==========+========+=========+====================================+
|        |     0 |        0 |      0 |       0 | Initial State                      |
+--------+-------+----------+--------+---------+------------------------------------+
| 1      |     1 |        0 |      1 |       0 | Man takes Goat to the right bank.  |
+--------+-------+----------+--------+---------+------------------------------------+
| 2      |     0 |        0 |      1 |       0 | Man crosses alone.                 |
+--------+-------+----------+--------+---------+------------------------------------+
| 3      |     1 |        0 |      1 |       1 | Man takes Grass to the right bank. |
+--------+-------+----------+--------+---------+------------------------------------+
| 4      |     0 |        0 |      0 |       1 | Man takes Goat to the left bank.   |
+--------+-------+----------+--------+---------+------------------------------------+
| 5      |     1 |        1 |      0 |       1 | Man takes Leopard to the right bank. |
+--------+-------+----------+--------+---------+------------------------------------+
| 6      |     0 |        1 |      0 |       1 | Man crosses alone.                 |
+--------+-------+----------+--------+---------+------------------------------------+
| 7      |     1 |        1 |      1 |       1 | Man takes Goat to the right bank.  |
+--------+-------+----------+--------+---------+------------------------------------+
|        |     1 |        1 |      1 |       1 | Reached goal state                 |
+--------+-------+----------+--------+---------+------------------------------------+
```

5. Comparing BFS and DFS Performance

Key Differences

Feature	BFS	DFS
Search Strategy	Explores all states level-by-level	Explores one deep path first
Optimality	Finds shortest path	May find longer paths
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$ (high memory)	O(d) (low memory)
Best Use Case	When shortest path is needed	When memory is limited

Execution Time Measurement

```
import time
from BFS import bfs
from DFS import dfs


# Measure BFS
```

```
start = time.time()
bfs_solution = bfs()
print(f"BFS Time: {time.time() - start:.6f} sec")

# Measure DFS
start = time.time()
dfs_solution = dfs()
print(f"DFS Time: {time.time() - start:.6f} sec")
```

Output

```
BFS Time: 0.000066 sec
DFS Time: 0.000052 sec
(env) → AI Assignment
```

Analysis:

- **BFS** explores all nodes at each level but guarantees the shortest path.
- **DFS** might reach the solution faster in small state spaces due to less overhead.

Solution Path Length Comparison

Algorithm	Solution Length
BFS	7 steps (shortest)
DFS	7 steps (same here)

BFS **always** guarantees the shortest path.
DFS **may** find the same path in simple puzzles but often explores redundant states in complex problems.

Memory Usage Comparison

Algorithm	Memory Usage
BFS	High
DFS	Low

BFS stores all nodes at each level (**exponential growth**).
DFS stores only the current path (**linear growth**).

Visualizing Paths

BFS Path (Level-by-Level Exploration)

(0,0,0,0) → (1,0,1,0) → (0,0,1,0) → (1,1,1,0) → (0,1,0,0) → (1,1,0,1) → (0,1,0,1) → (1,1,1,1)

DFS Path (Depth-First Search Path)
(0,0,0,0) → (1,0,1,0) → (0,0,1,0) → (1,0,1,1) → (0,0,0,1) → (1,1,0,1) → (0,1,0,1) → (1,1,1,1)

Same as BFS in this case due to limited state space.

Final Verdict

BFS is the better choice for this puzzle

- Guarantees the **shortest path**.
- **Memory is manageable** due to the small state space.

DFS is riskier

- Uses **less memory** but may explore **longer paths** in complex problems.
- Works fine here, but **not ideal for larger state spaces**.

6. Conclusion

Key Takeaways

- **BFS is the best choice** as it finds the shortest solution.
- **DFS is useful** but can lead to longer or looping paths.
- **Graph representation** helps visualize and debug the problem-solving process.

By implementing BFS and DFS, we efficiently solve the **river-crossing puzzle** while ensuring safe transitions between states.