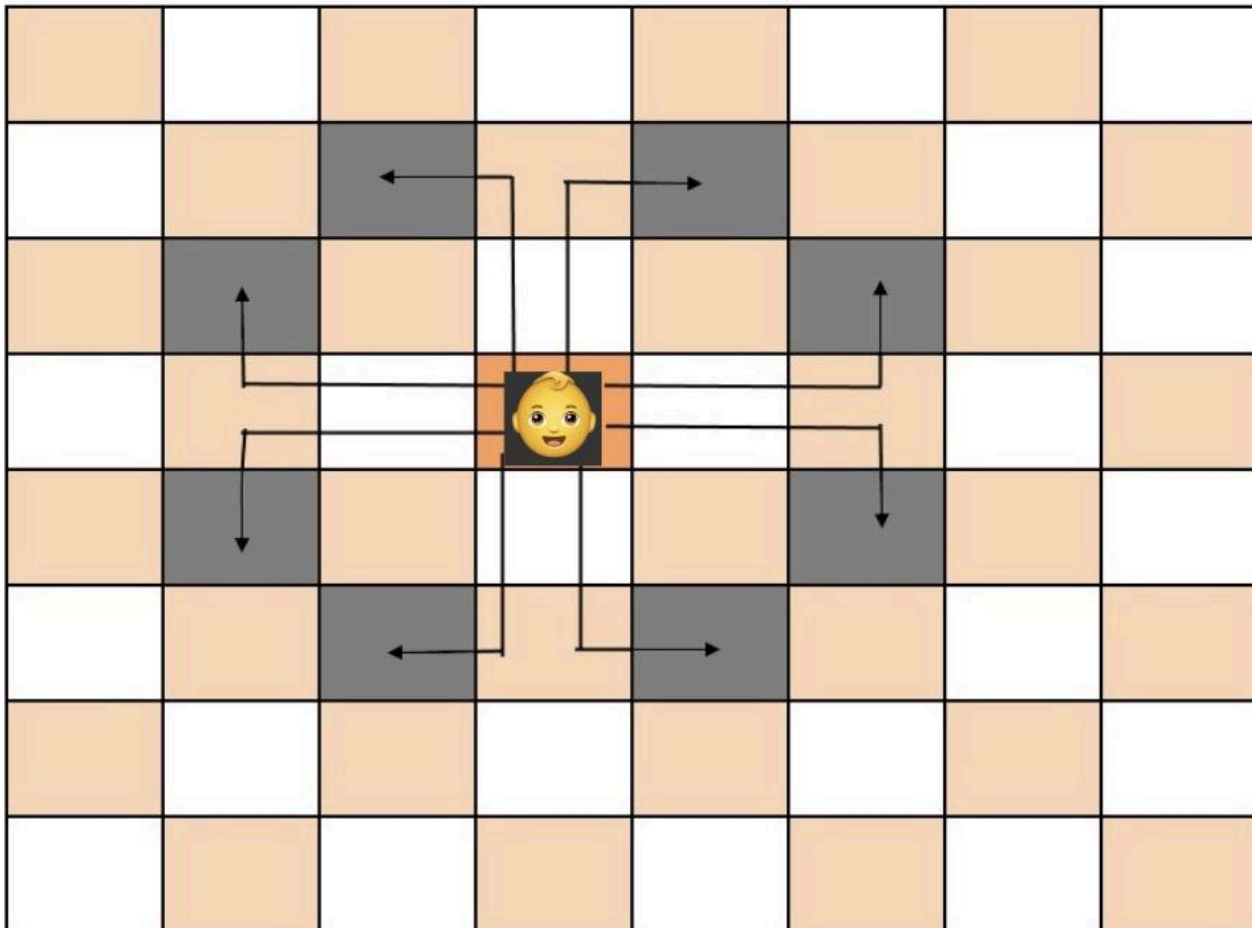


Search Methods in AI

Toddler Toy Trek

Problem Description

In a magical 8x8 play area, a little boy and his pet cat are on a thrilling adventure to collect all the toys scattered across the grid. However, there's a catch! The cat, mischievous and playful, can move on the grid too, but unlike the boy, it moves like a rook in chess (horizontally or vertically).

The boy must move only in an L shape and collect all the toys to win the game. But he must be careful! If the cat lands on the same square as the boy, the game will be over.

Your mission is to help the boy navigate the play area, gather all the toys, and avoid losing to the cat.

Why is this problem interesting?

This problem is interesting because it builds on the classical Knight's Tour by adding new constraints. The boy moves like a knight, but he must avoid the cat, which moves like a rook, and collect toys on the grid. The dynamic interaction between the boy and the cat adds real-time decision-making complexity. The challenge lies in balancing two goals—avoiding the cat and collecting toys—while designing effective search algorithms and heuristics to solve the problem efficiently. This makes it a rich exploration of state space search, multi-objective optimization, and AI planning.

State Representation:

The state is represented as a tuple (bx, by, cx, cy, toys_collected), where:

- (bx, by) represents the boy's current position on the 8x8 grid.
- (cx, cy) represents the cat's current position on the 8x8 grid.
- toys_collected is an 8x8 binary matrix, where:
- toys_collected[i][j] = 1 indicates that the toy on tile (i, j) has been collected.
- toys_collected[i][j] = 0 indicates that the toy on tile (i, j) is still there.

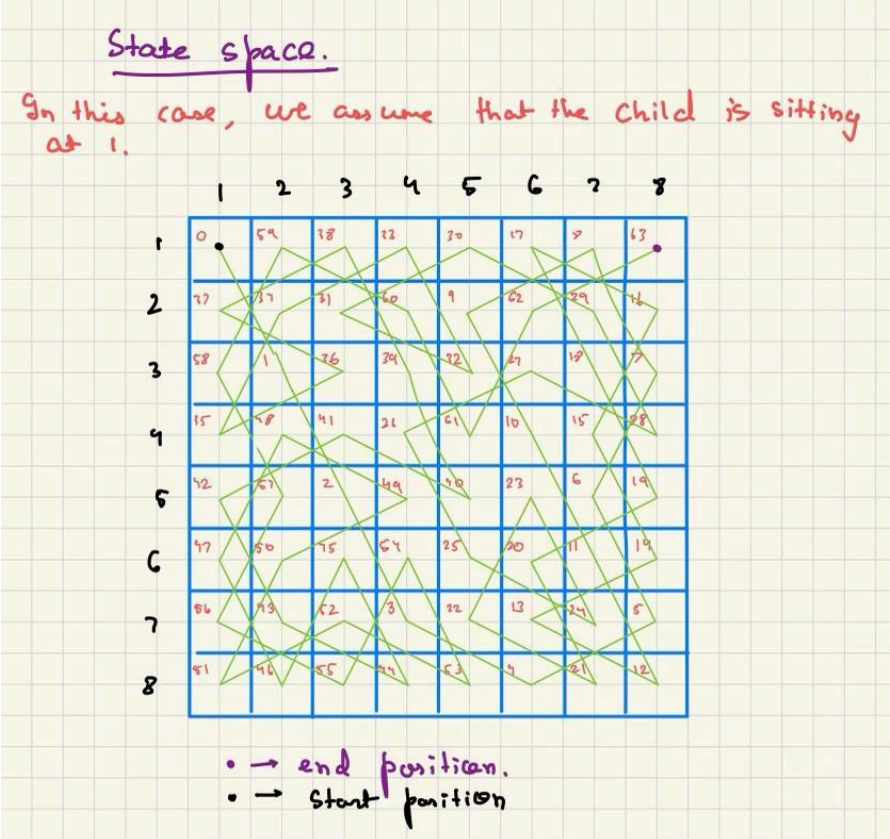
Movement Rules

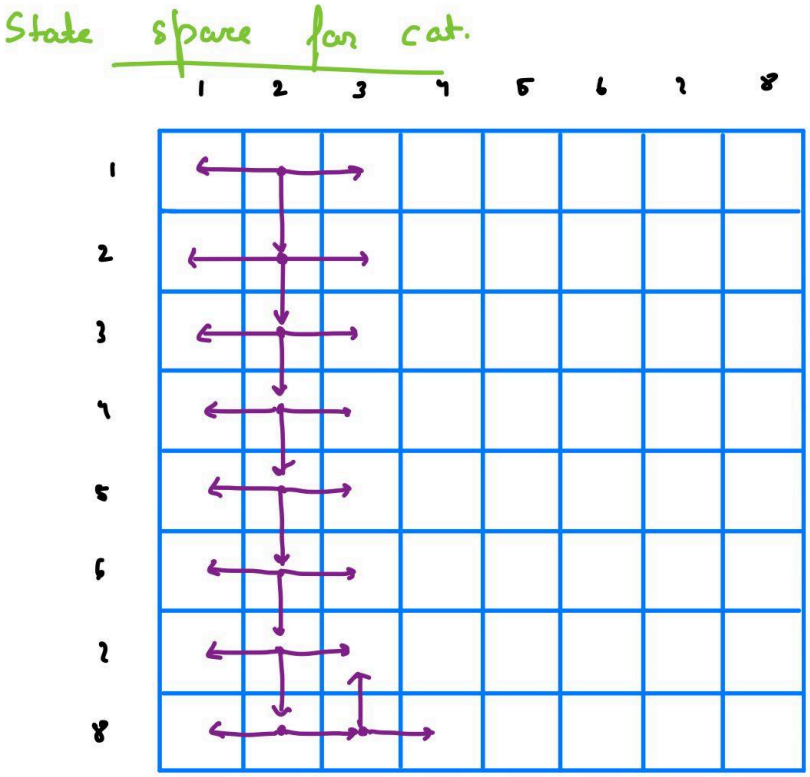
Boy's Movement:

- The boy moves like a knight in chess: two squares in one direction and one square in a perpendicular direction.
- The boy does not revisit any visited cell.

Cat's Movement:

- The cat moves like a rook in chess: it can move one square either horizontally or vertically(up, down, left, right).
- The cat's movement is determined by a simple rule: it always moves toward the boy's current position, but it cannot move diagonally.
- The cat moves after the boy, and if it reaches the boy's position, the game ends.

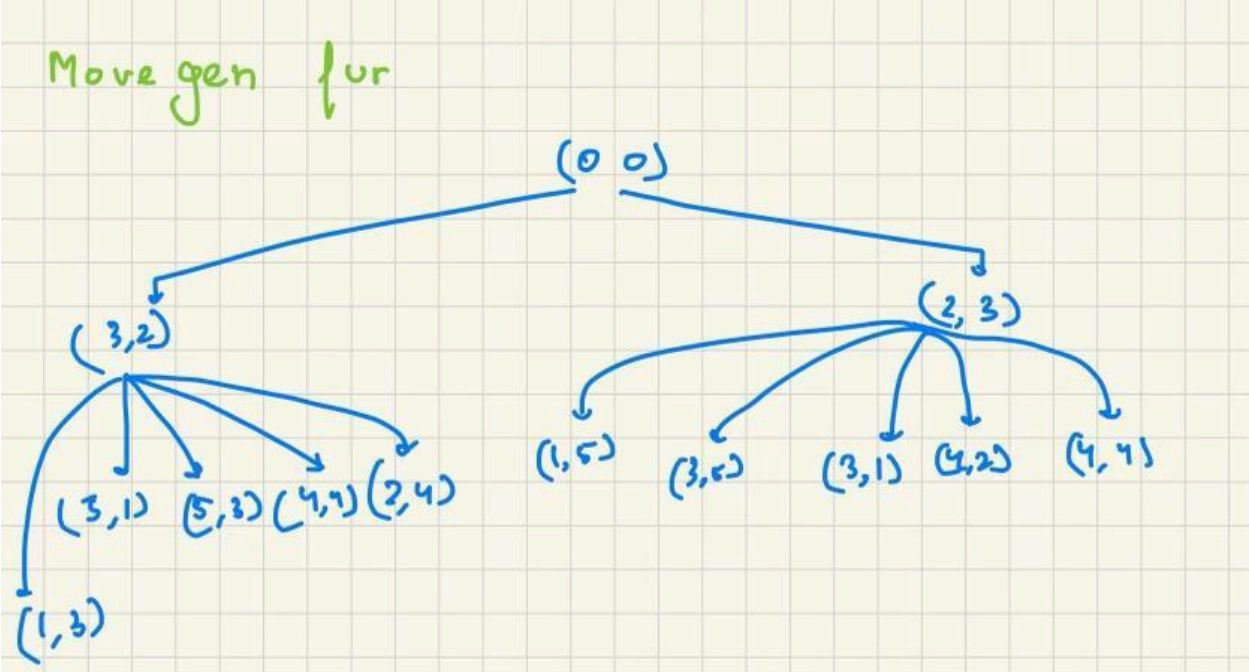

State space.
In this case, we assume that the child is sitting at 1.
1 2 3 4 5 6 7 8
1
2
3
4
5
6
7
8
• → end position.
• → start position



MoveGen Description

The MoveGen function takes the current state as input and returns a set of valid next states for both the boy and the cat. The function considers all possible L-shaped moves for the boy and filters out invalid moves that either revisit a square or move outside the grid. The cat then moves toward the boy's new position.

- The function calculates all possible L-shaped moves from (x,y) according to the child's movement on the grid.
- It then filters out any moves that are outside the grid or lead to a square that has already been visited (i.e., where toys_collected[x′][y′]=1).




```python
def MoveGen(n, bx, by, cx, cy, toys_collected):

    boy_dx = [2, 2, 1, 1, -1, -1, -2, -2]

    boy_dy = [1, -1, 2, -2, 2, -2, 1, -1]



    valid_moves = []

    for i in range(8):

        new_bx = bx + boy_dx[i]

        new_by = by + boy_dy[i]



        # Ensure the move is within bounds and the toy hasn't been
collected

        if 1 <= new_bx <= n and 1 <= new_by <= n and toys_collected[new_bx
- 1][new_by - 1] == 0:
```

```
            # Move the cat toward the new boy's position
            new_cx, new_cy = MoveCat(cx, cy, new_bx, new_by)
            # Ensure the boy and cat don't collide
            if (new_bx, new_by) != (new_cx, new_cy):
                new_toys_collected = [row[:] for row in toys_collected]
                new_toys_collected[new_bx - 1][new_by - 1] = 1
                valid_moves.append((new_bx, new_by, new_cx, new_cy,
new_toys_collected))

    return valid_moves

def MoveCat(cx, cy, bx, by):
    # Move cat toward the boy's position
    if cx < bx:
        cx += 1
    elif cx > bx:
        cx -= 1
    if cy < by:
        cy += 1
    elif cy > by:
        cy -= 1
    return cx, cy
```

GoalTest

- The GoalTest function checks whether the boy has collected all the toys. The game is won if every entry in the toys_collected matrix is 1. The game ends if the boy and the cat occupy the same square.

```
def GoalTest(bx, by, cx, cy, toys_collected):
    if (bx, by) == (cx, cy):
        return False  # Game over if boy and cat collide
    for row in toys_collected:
        if 0 in row:
            return False  # Toys are still remaining
    return True  # All toys collected
```

User Input for Start State and Goal State:

To allow users to specify the start state and goal state, we can create a simple interface:

1. For the start state:
    - Ask the user for the grid dimensions (rows and columns)
    - Ask for the boy's and cat's starting positions.

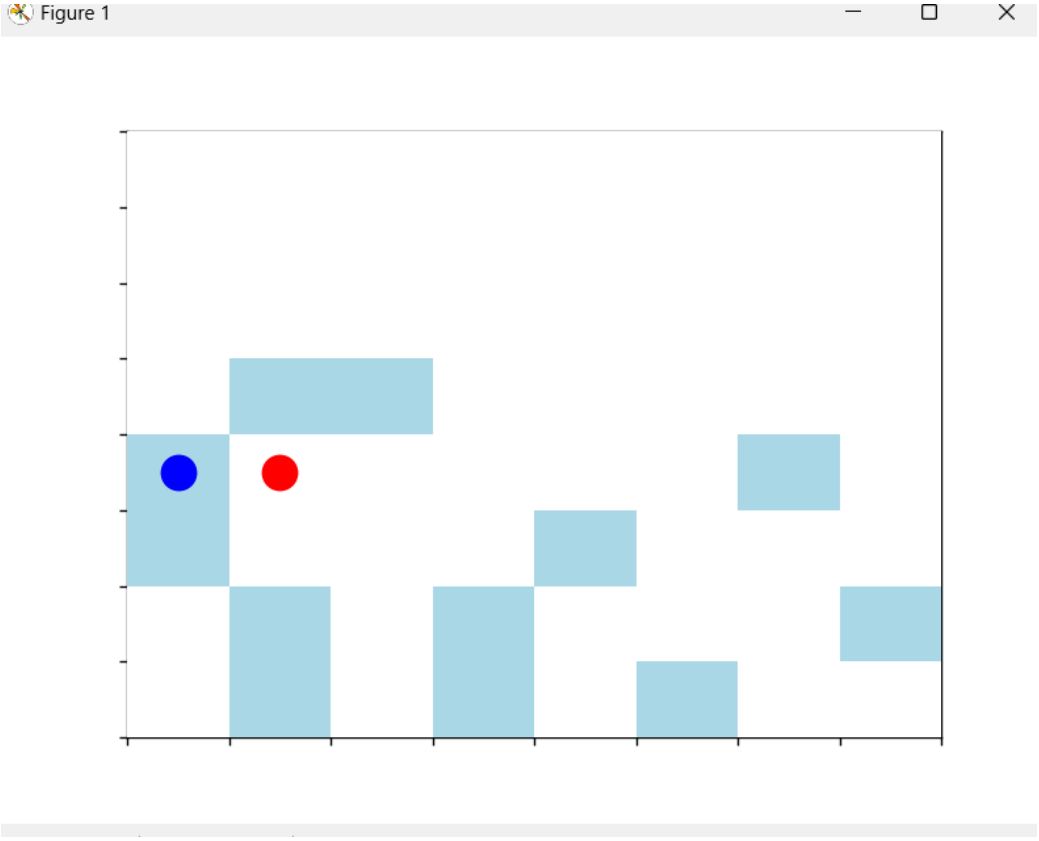
Output:

```
Solution found with boy starting at (5, 2) and cat starting at (6, 3)!
Boy's path: [(5, 2), (3, 1), (1, 2), (2, 4), (1, 2), (2, 4), (1, 6), (2, 8), (4, 7), (2, 8), (4, 7), (3, 5), (1, 4), (2, 2), (4, 1),
(5, 3), (4, 1), (2, 2), (1, 4), (2, 6), (1, 8), (3, 7), (1, 8), (3, 7), (2, 5), (1, 3), (2, 1), (4, 2), (2, 1), (4, 2), (3, 4), (1, 3
), (2, 5), (1, 7), (3, 6), (2, 4), (1, 2), (3, 1), (4, 3), (3, 1), (5, 2), (4, 4), (2, 3), (1, 5), (2, 7), (4, 6), (3, 4), (1, 3), (3
, 2), (5, 1), (6, 3), (5, 1), (3, 2), (1, 1), (2, 3), (1, 5), (2, 7), (4, 6), (2, 7), (4, 8), (5, 6), (4, 4), (2, 3), (1, 5), (2, 3),
 (1, 1), (3, 2), (4, 4), (3, 6), (1, 7), (3, 8), (5, 7), (4, 5), (2, 4), (3, 2), (5, 1), (7, 2), (6, 4), (4, 3), (2, 2), (1, 4), (3,
5), (1, 4), (2, 2), (4, 1), (6, 2), (5, 4), (3, 3), (1, 4), (2, 6), (4, 5), (3, 3), (4, 1), (6, 2), (7, 4), (5, 5), (3, 4), (2, 6), (
4, 7), (6, 6), (5, 4), (3, 3), (2, 5), (3, 7), (5, 8), (7, 7), (6, 5), (4, 4), (2, 3), (4, 2), (6, 1), (7, 3), (6, 1), (4, 2), (2, 1)
```

Implementation with DFS

Code : https://github.com/vijeraghu/Search-Methods-In-AI

It was successful in finding solutions for specific start positions of the cat and the boy.



Implementation with BFS

Code : https://github.com/vijeraghu/Search-Methods-In-AI

Our implementation with BFS, however, did not converge even after 100000 iterations, it was around this point that we started running out of memory and could not find a convergence for the BFS solution, even with optimizations like making the boy not visit the cat's positions and removing any visited nodes, we were unable to make it converge.

```
Iteration 98000, Queue size: 221786, Visited states: 88081
Iteration 99000, Queue size: 223883, Visited states: 88991
Iteration 100000, Queue size: 226138, Visited states: 89924
No solution found after 100000 iterations
No solution found.
```

Heuristic Function:

The goal of the heuristic is to: Best First Search

Code : https://github.com/vijeraghu/Search-Methods-In-AI

1. **Maximize the distance from the cat**: This is to avoid a collision with the cat, which is critical to survival.
2. **Minimize the distance to the nearest uncollected toy**: This is to prioritize collecting toys, as the boy needs to collect all toys to achieve the goal.

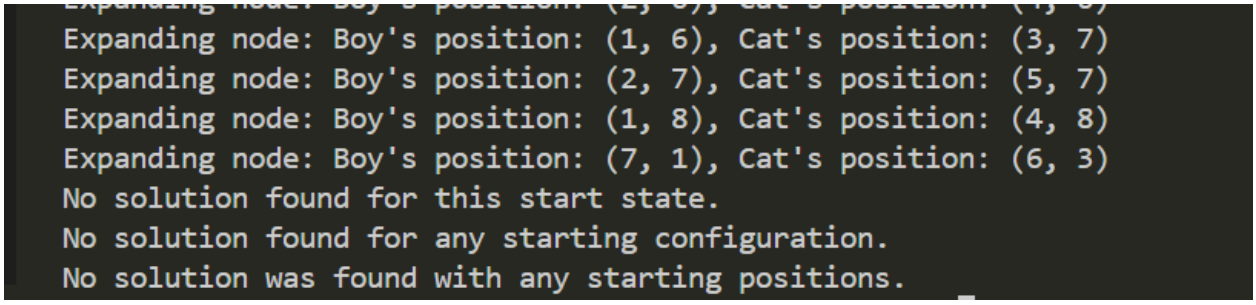
The heuristic function h(bx,by,cx,cy,toys_collected):

h(bx,by,cx,cy,toys_collected)= dist_to_nearest_toy(bx,by,toys_collected)− dist_to_cat(bx,by,cx,cy)

Where:

- dist_to_nearest_toy(bx,by,toys_collected): The Manhattan distance from the boy's position (bx,by) to the nearest uncollected toy.

- dist_to_cat(bx,by,cx,cy): The Manhattan distance from the boy's position (bx,by) to the cat's position (cx,cy).



Implementation Report

Depth First Search (DFS) DFS explores as far as possible along each branch before backtracking. It uses a stack to manage the exploration:

Performance: DFS successfully found a solution for the specific start positions of the boy and the cat.

Breadth First Search (BFS) BFS explores all nodes at the present depth level before moving to the next level. It uses a queue to manage exploration:

Performance: BFS did not find a solution within the given constraints. The number of iterations exceeded practical limits, likely due to the large search space and high memory consumption required to store nodes at each depth level.

Best First Search (Heuristic-Based Search) Best First Search uses a heuristic to prioritize nodes closer to the goal. In this case, the Manhattan Distance heuristic was employed:

Performance: The algorithm did not find a solution and experienced a timeout. This may be due to several factors, including the heuristic's accuracy and the complexity of navigating the grid.

Performance Evaluation

DFS Performance

Time Taken: Variable; dependent on the depth of the solution path.

Nodes Explored: Can be high if deep branches are explored.

Memory Usage: Low, as it only needs to store the path from the root to the current node.
Note: DFS can find The solution if and only if the boy is allowed to revisit the cells.

BFS Performance

Time Taken: High, due to exploration of all nodes at each depth level.

Nodes Explored: Potentially very high, especially on larger grids.

Memory Usage: High, as it stores all nodes at the current depth level.

Best First Search Performance

Could not find any solution for any of the starting state configurations.The search space increases exponentially to find out the path in case repetition is allowed and if the repetition is not allowed solution for any start pair of the boy and cat is found.

Nodes Explored: High, as the heuristic may not effectively guide the search.

Memory Usage: High, depending on the heuristic's nature accross the grid.