Date 25/10/2024
Page

Lab-3

⇒ A* Algorithm

function A* search (problem) returns a solution or failure
node ← a node n, with n-state-problem.initial state n.g = 0
frontier ← a priority queue ordered by ascending g/h, only elements

loop do
if empty? (frontier) then return failure
n ← pop (frontier)
if problem.goalTest (n.state) then return solution(n)
for each action a in problem.action (n.state) do
n ← childNode (problem, n, a)
insert (n', g(n') + h(n'), frontier)

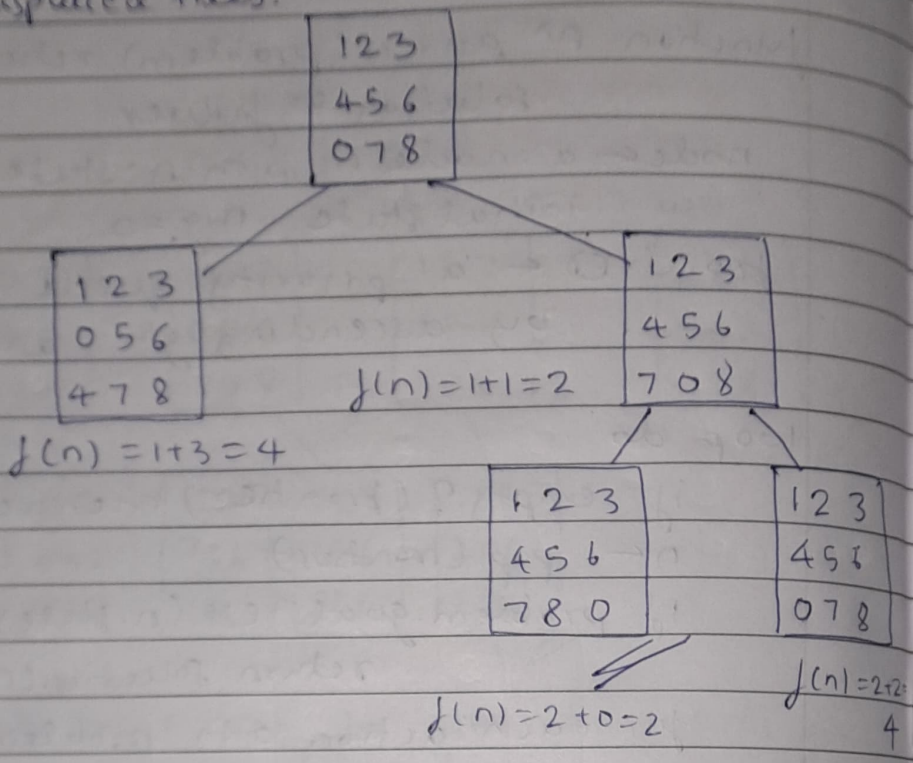
Misplaced tiles
f(n) = g(n) + h(n)
g(n) → depth
h(n) → no. of misplaced tiles

Manhattan distance
f(n) = sum of distance of each tile from current position to goal state

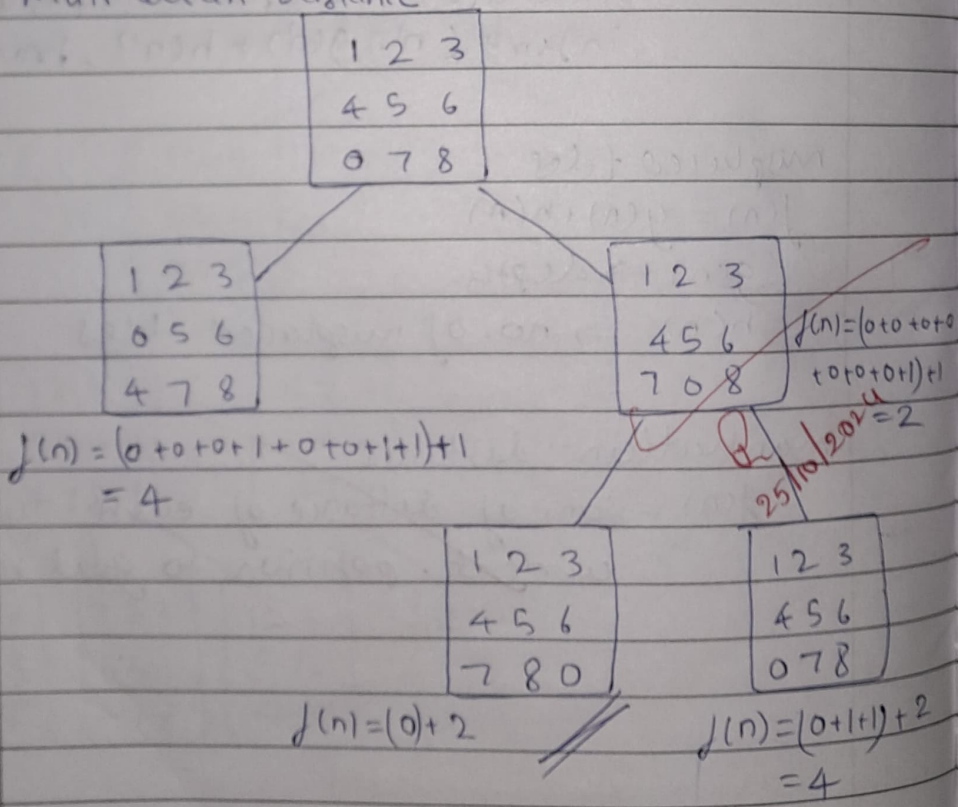
Date __/__/__
Page

* State space tree

→ Misplaced tiles.



→ Manhattan distance



astar

November 9, 2024

[2]:
```
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")
import heapq

class PuzzleState:
    def __init__(self, board, g=0):
        self.board = board
        self.g = g
        self.zero_pos = board.index(0)

    def h(self):
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] !
 ↪= i + 1) #misplaced tiles

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]

                new_board[self.zero_pos], new_board[new_zero_pos] =␣
 ↪new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))
    came_from = {}
    g_score = {tuple(initial_state.board): 0}
```

```
    while open_set:
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1

            if neighbor_tuple not in g_score or tentative_g_score <␣
 ↪g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor)) ␣
 ↪# Use neighbor.g as the tie-breaker

    return None

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = a_star(initial_state, goal_state)

if solution:
    for step in solution:
        print(step)
else:
    print("No solution found")
```

```
Name:Sudarshan Komar
USN:1BM22CS291
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
```

[3]:
```
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")
import heapq

class PuzzleState:
```

```
    def __init__(self, board):
        self.board = board
        self.zero_pos = board.index(0)

    def h(self):
        distance = 0
        for i in range(9):
            if self.board[i] != 0:
                target_x, target_y = divmod(self.board[i] - 1, 3)
                current_x, current_y = divmod(i, 3)
                distance += abs(target_x - current_x) + abs(target_y -␣
 ↪current_y)
        return distance

    def f(self):
        return self.h()  # Just the heuristic value (Manhattan distance)

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                new_board[self.zero_pos], new_board[new_zero_pos] =␣
 ↪new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), id(initial_state),␣
 ↪initial_state))
    came_from = {}
    g_score = {tuple(initial_state.board): 0}

    while open_set:
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
```

```
            tentative_g_score = g_score[tuple(current.board)] + 1  # All edges␣
 ↪have a cost of 1

            if neighbor_tuple not in g_score or tentative_g_score <␣
 ↪g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                heapq.heappush(open_set, (tentative_g_score + neighbor.h(),␣
 ↪id(neighbor), neighbor))

    return None

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = a_star(initial_state, goal_state)

if solution:
    for step in solution:
        print(step)
else:
    print("No solution found")
```

```
Name:Sudarshan Komar
USN:1BM22CS291
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
```