AI Lab Assignment

CSE-4132

Implementing Searching Techniques & Algorithms in AI

Heuristic Search
Best-First Search
A* Search

Submitted by

Group: 8

Omor Faruk Niloy
ID: 2010376103

Mst. Shimla Sinthia
ID: 2012376105

Nourin Nusrat Jhahan
ID: 2012476112

MD. Farhan Yousuf
ID: 2010276139

Nusrat Jannat
ID: 2012176145

Submitted To

DR. A. K. M. AKHTAR HOSSAIN
Professor
Computer Science and Engineering
University of Rajshahi

Implementing Heuristic Search to solve 8-puzzle problem

import library

In [ ]:
```
import heapq
import copy
```

Create node class

In [ ]:
```
class PuzzleNode:
  def __init__(self, state, parent=None, move=None, g=0, h=0):
      self.state = state
      self.parent = parent
      self.move = move
      self.g = g
      self.h = h
      self.f = g + h

  def __lt__(self, other):
      return self.h < other.h
```

printing functions

In [ ]:
```
def print_state(node):
    state = node.state
    g_str = f"g={node.g}"
    h_str = f"h={node.h}"
    f_str = f"f={node.f}"

    value_lines = [g_str, h_str, f_str]

    for i, row in enumerate(state):
        row_str = " ".join(str(num) if num != 0 else " " for num in row)
        value_display = value_lines[i] if i < len(value_lines) else ""
        print(f"{row_str}   {value_display}")

    print("-" * 15)
```

In [ ]:
```
def print_states_with_values(nodes):
    states = [node.state for node in nodes]
    moves = [node.move if node.move else "Start" for node in nodes]
    g_values = [f"g={node.g}" for node in nodes]
    h_values = [f"h={node.h}" for node in nodes]
    f_values = [f"f={node.f}" for node in nodes]

    print("  ".join(f"{move:^6}" for move in moves))

    for row in range(3):
        print("    ".join(" ".join(str(num) if num != 0 else " "
for num in state[row]) for state in states))

    print("     ".join(g_values))
    print("     ".join(h_values))
    print("     ".join(f_values))
    print("-" * (len(states) * 22))
```

Path reconstruction

In [ ]:
```
def reconstruct_path(node):
    path = []
    while node.parent:
        path.append(node.move)
```

```
        node = node.parent # backtrack to the parent node
    return path[::-1]
```

Heuristic funtion

In [ ]:
```
def heuristic_function(state, goal):
    h = 0
    for i in range(3):
        for j in range(3):
            if (goal[i][j] == '*'):
                continue
            # check if the cell doesn't match with the goal
            if state[i][j] != goal[i][j]:
                h += 1
    return h
```

Make hashvalue for a node to insert into visited set

In [ ]:
```
def flatten(state):
    return tuple(sum(state, []))
```

Generate neighbors

In [ ]:
```
def get_neighbors(state, visited):
    neighbors = []
    moves = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}

    zero_x = -1
    zero_y = -1
    # finding empty cell
    for i in range(3):
        for j in range(3):
            if state[i][j] == '*':
                zero_x, zero_y = i, j
                break

    #iterate in four direction of empty cell
    for move, (dx, dy) in moves.items():
        new_x, new_y = zero_x + dx, zero_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = copy.deepcopy(state)
            new_state[zero_x][zero_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[zero_x][zero_y]

            # check if the state is already visited or not
            if flatten(new_state) in visited:
                continue
            neighbors.append((new_state, move))

    return neighbors
```

Heuristic search

In [ ]:
```
def heuristic_search(start, goal):
    pq = [] # priority queue to give priority to lowest heuristic value state
    heapq.heappush(pq, PuzzleNode(start, None, None, 0, heuristic_function(start, goal)))
    visited = set() # a set of already visited states

    while pq:
        current_node = heapq.heappop(pq)
        print("Current State:")
        print_state(current_node)

        if current_node.state == goal: # checking if the current state is the goal state or not
            return reconstruct_path(current_node)
        visited.add(flatten(current_node.state)) # mark current state as a visited state

        #generate neighbors
        neighbors = get_neighbors(current_node.state, visited)
```

```
        neighbor_nodes = []
        for neighbor, move in neighbors:
            g = current_node.g + 1
            h = heuristic_function(neighbor, goal)
            heapq.heappush(pq, PuzzleNode(neighbor, current_node, move, g, h))
            neighbor_nodes.append(PuzzleNode(neighbor, current_node, move, g, h))

        # print neighbors
        if neighbor_nodes:
            print("Generated Neighbors:")
            print_states_with_values(neighbor_nodes)
            print('\n')
    return None
```

Make start and goal

In [ ]:
```
start_state = [['2', '8', '3'], ['1', '6', '4'], ['7', '*', '5']]  # initial state
goal_state = [['1', '2', '3'], ['8', '*', '4'], ['7', '6', '5']]   # Goal state

solution = heuristic_search(start_state, goal_state)
if solution:
    print("Solution found:", solution)
else:
    print("No solution found.")
```

```
Current State:
2 8 3   g=0
1 6 4   h=4
7 * 5   f=4
---------------
Generated Neighbors:
  U       L       R
2 8 3   2 8 3   2 8 3
1 * 4   1 6 4   1 6 4
7 6 5   * 7 5   7 5 *
g=1     g=1     g=1
h=3     h=5     h=5
f=4     f=6     f=6
-------------------------------------------------------------------


Current State:
2 8 3   g=1
1 * 4   h=3
7 6 5   f=4
---------------
Generated Neighbors:
  U       L       R
2 * 3   2 8 3   2 8 3
1 8 4   * 1 4   1 4 *
7 6 5   7 6 5   7 6 5
g=2     g=2     g=2
h=3     h=3     h=4
f=5     f=5     f=6
-------------------------------------------------------------------


Current State:
2 * 3   g=2
1 8 4   h=3
7 6 5   f=5
---------------
Generated Neighbors:
  L       R
* 2 3   2 3 *
1 8 4   1 8 4
7 6 5   7 6 5
g=3     g=3
h=2     h=4
f=5     f=7
---------------------------------------------


Current State:
* 2 3   g=3
1 8 4   h=2
7 6 5   f=5
---------------
Generated Neighbors:
  D
1 2 3
* 8 4
7 6 5
g=4
h=1
f=5
----------------------


Current State:
1 2 3   g=4
* 8 4   h=1
7 6 5   f=5
---------------
Generated Neighbors:
  D       R
1 2 3   1 2 3
7 8 4   8 * 4
* 6 5   7 6 5
g=5     g=5
h=2     h=0
f=7     f=5
---------------------------------------------
```

```
Current State:
1 2 3   g=5
8 * 4   h=0
7 6 5   f=5
---------------
Solution found: ['U', 'U', 'L', 'D', 'R']
```

Implementing Best-First Search (BFS) Algorithm

Import Library

In [ ]:
```
import heapq
```

Creating The Graph Class(Including add_edge and best_first_search function)

In [ ]:
```
class BestFirstSearch:
    def __init__(self):
        self.graph = {}
        self.heuristic = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def set_heuristic(self, heuristic):
        self.heuristic = heuristic

    def best_first_search(self, start, goal):
        open_list = []   # Priority queue
        close_list = []  # List to track expanded nodes
        parent = {}      # Store parent nodes to reconstruct the path
        heapq.heappush(open_list, (self.heuristic[start], start))
        parent[start] = None

        print(f"Initialization:")
        print(f"Open List: {[node for _, node in open_list]}")
        print(f"Close List: {close_list}\n")

        while open_list:
            _, current = heapq.heappop(open_list)
            close_list.append(current)

            print(f"Expanding Node: {current}")
            print(f"Close List: {close_list}")

            if current == goal:
                print("\nGoal Reached!")
                path = []
                while current is not None:
                    path.append(current)
                    current = parent[current]
                path.reverse()
                print(f"Final Path: {' → '.join(path)}")
                return

            for neighbor in self.graph.get(current, []):
                if neighbor not in close_list:
                    # Check if the neighbor is already in the open list
                    if not any(n[1] == neighbor for n in open_list):
                        heapq.heappush(open_list, (self.heuristic[neighbor], neighbor))
                        parent[neighbor] = current

            # Sort open list by heuristic to ensure priority queue order
            open_list.sort(key=lambda x: x[0])

            print(f"Open List: {[node for _, node in open_list]}\n")

        print("Goal not reachable.")
```

Adding edges

In [ ]:
```
if __name__ == "__main__":
    bfs = BestFirstSearch()

    # Adding edges
    bfs.add_edge('S', 'A')
    bfs.add_edge('S', 'B')
    bfs.add_edge('A', 'C')
    bfs.add_edge('A', 'D')
    bfs.add_edge('B', 'E')
    bfs.add_edge('B', 'F')
    bfs.add_edge('E', 'H')
    bfs.add_edge('E', 'I')
    bfs.add_edge('F', 'G')  # Goal node

    # Setting heuristic values
    heuristic_values = {
        'S': 14, 'A': 12, 'B': 5, 'C': 7, 'D': 3, 'E': 8,
        'F': 2, 'H': 4, 'I': 9, 'G': 0
    }
    bfs.set_heuristic(heuristic_values)

    # Running Best-First Search
    bfs.best_first_search('S', 'G')
```

```
Initialization:
Open List: ['S']
Close List: []

Expanding Node: S
Close List: ['S']
Open List: ['B', 'A']

Expanding Node: B
Close List: ['S', 'B']
Open List: ['F', 'E', 'A']

Expanding Node: F
Close List: ['S', 'B', 'F']
Open List: ['G', 'E', 'A']

Expanding Node: G
Close List: ['S', 'B', 'F', 'G']

Goal Reached!
Final Path: S → B → F → G
```

Implementing A* Search Algorithm

Import Library

In [ ]:
```
import heapq
```

Creating The Graph Class(Including add_edge and a_star_search function)

In [ ]:
```
class Graph:
    def __init__(self):
        self.graph = {}
    # Adding edges
    def add_edge(self, u, v, cost):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append((v, cost))

    # A* Search Algorithm
    def a_star_search(self, start, goal, heuristic):
        pq = []

        # (f(n) = g(n) + h(n), g(n), node)
        heapq.heappush(pq, (heuristic[start], 0, start))

        g_costs = {start: 0}  # storing the actual costs (g(n))
        parent = {start: None}  # tracking the shortest path

        while pq:
            # Taking the node with the lowest f(n)
            f_cost, g_cost, node = heapq.heappop(pq)
            if node == goal:
                path = []
                while node:
                    path.append(node)
                    node = parent[node]
                path.reverse()
                print("Shortest Path:", " -> ".join(path))
                print("Total Cost:", g_costs[goal])
                return

            for neighbor, edge_cost in self.graph.get(node, []):
                new_g_cost = g_cost + edge_cost

                # If new g(n) is better then the current or neighbor is not yet visited
                if neighbor not in g_costs or new_g_cost < g_costs[neighbor]:
                    g_costs[neighbor] = new_g_cost
                    f_cost = new_g_cost + heuristic[neighbor]
                    heapq.heappush(pq, (f_cost, new_g_cost, neighbor))
                    parent[neighbor] = node

        print("No path found from", start, "to", goal) # If the goal is unreachable
```

Taking The Input

In [ ]:
```
# Creating the graph
graph = Graph()

# Static Input for edges
edges = [
    ("S", "A", 1),
    ("S", "G", 10),
    ("A", "C", 1),
    ("A", "B", 2),
    ("C", "D", 3),
    ("C", "G", 4),
    ("B", "D", 5),
```

```python
    ("D", "G", 2)
]

# Adding edges
for u, v, cost in edges:
    graph.add_edge(u, v, cost)

# Given heuristic values
heuristic = {
    "S": 5,
    "A": 3,
    "B": 4,
    "C": 2,
    "D": 6,
    "G": 0
}

# Start and goal nodes
start = "S"
goal = "G"

print("Start of Searching " + start)
print("Goal of Searching " + goal)
print("\nExecuting A* Search...")
graph.a_star_search(start, goal, heuristic)
```

```
Start of Searching S
Goal of Searching G

Executing A* Search...
Shortest Path: S -> A -> C -> G
Total Cost: 6
```