Samkit Shah Bhavya Shah

2019130060, 2019130056

TE Comps

Batch C

Experiment – 2

Aim: To implement the fire extinguisher system using BFS and DFS.

Theory:

- **Depth-First-Search (DFS)**:
    1. DFS always expands DEPTH-FIRST the deepest node in the current frontier of the search tree.
    2. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
    3. DFS uses a LIFO queue.
    4. Visits children before siblings.

- **Breadth-First-Search (BFS):**

1. BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
2. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
3. BFS uses a FIFO queue.
4. Visits siblings before children

Problem Statement : Solving 8 puzzle problem using dfs and bfs algorithm and comparing both the algorithms.

Code:

```
class State:

   def __init__(self, state, parent, move, depth, cost, key):

       self.state = state

       self.parent = parent

       self.move = move

       self.depth = depth

       self.cost = cost

       self.key = key
```

```python
        if self.state:
            self.map = ''.join(str(e) for e in self.state)

    def __eq__(self, other):
        return self.map == other.map

    def __lt__(self, other):
        return self.map < other.map


import argparse
import timeit
import resource
from collections import deque
from heapq import heappush, heappop, heapify
import itertools

goal_state = [0, 1, 2, 3, 4, 5, 6, 7, 8]
goal_node = State
initial_state = list()
board_len = 0
board_side = 0

nodes_expanded = 0
max_search_depth = 0
max_frontier_size = 0

moves = list()
costs = set()


def bfs(start_state):

    global max_frontier_size, goal_node, max_search_depth

    explored, queue = set(), deque([State(start_state, None, None, 0, 0, 0)])

    while queue:

        node = queue.popleft()
```

```python
        explored.add(node.map)

        if node.state == goal_state:
            goal_node = node
            return queue

        neighbors = expand(node)

        for neighbor in neighbors:
            if neighbor.map not in explored:
                queue.append(neighbor)
                explored.add(neighbor.map)

                if neighbor.depth > max_search_depth:
                    max_search_depth += 1

        if len(queue) > max_frontier_size:
            max_frontier_size = len(queue)


def dfs(start_state):

    global max_frontier_size, goal_node, max_search_depth

    explored, stack = set(), list([State(start_state, None, None, 0, 0, 0)])

    while stack:

        node = stack.pop()

        explored.add(node.map)

        if node.state == goal_state:
            goal_node = node
            return stack

        neighbors = reversed(expand(node))

        for neighbor in neighbors:
            if neighbor.map not in explored:
                stack.append(neighbor)
                explored.add(neighbor.map)
```

```
            if neighbor.depth > max_search_depth:
                max_search_depth += 1

        if len(stack) > max_frontier_size:
            max_frontier_size = len(stack)


def ast(start_state):

    global max_frontier_size, goal_node, max_search_depth

    explored, heap, heap_entry, counter = set(), list(), {}, itertools.count()

    key = h(start_state)

    root = State(start_state, None, None, 0, 0, key)

    entry = (key, 0, root)

    heappush(heap, entry)

    heap_entry[root.map] = entry

    while heap:

        node = heappop(heap)

        explored.add(node[2].map)

        if node[2].state == goal_state:
            goal_node = node[2]
            return heap

        neighbors = expand(node[2])

        for neighbor in neighbors:

            neighbor.key = neighbor.cost + h(neighbor.state)

            entry = (neighbor.key, neighbor.move, neighbor)
```

```
            if neighbor.map not in explored:

                heappush(heap, entry)

                explored.add(neighbor.map)

                heap_entry[neighbor.map] = entry

                if neighbor.depth > max_search_depth:
                    max_search_depth += 1

            elif neighbor.map in heap_entry and neighbor.key <
heap_entry[neighbor.map][2].key:

                hindex = heap.index((heap_entry[neighbor.map][2].key,
                                     heap_entry[neighbor.map][2].move,
                                     heap_entry[neighbor.map][2]))

                heap[int(hindex)] = entry

                heap_entry[neighbor.map] = entry

                heapify(heap)

        if len(heap) > max_frontier_size:
            max_frontier_size = len(heap)

def expand(node):

    global nodes_expanded
    nodes_expanded += 1

    neighbors = list()

    neighbors.append(State(move(node.state, 1), node, 1, node.depth + 1, node.cost +
1, 0))
    neighbors.append(State(move(node.state, 2), node, 2, node.depth + 1, node.cost +
1, 0))
    neighbors.append(State(move(node.state, 3), node, 3, node.depth + 1, node.cost +
1, 0))
    neighbors.append(State(move(node.state, 4), node, 4, node.depth + 1, node.cost +
1, 0))
```

```python
    nodes = [neighbor for neighbor in neighbors if neighbor.state]

    return nodes


def move(state, position):

    new_state = state[:]

    index = new_state.index(0)

    if position == 1:  # Up

        if index not in range(0, board_side):

            temp = new_state[index - board_side]
            new_state[index - board_side] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 2:  # Down

        if index not in range(board_len - board_side, board_len):

            temp = new_state[index + board_side]
            new_state[index + board_side] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 3:  # Left

        if index not in range(0, board_len, board_side):

            temp = new_state[index - 1]
            new_state[index - 1] = new_state[index]
```

```python
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 4:  # Right

        if index not in range(board_side - 1, board_len, board_side):

            temp = new_state[index + 1]
            new_state[index + 1] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None


def h(state):

    return sum(abs(b % board_side - g % board_side) + abs(b//board_side -
g//board_side)
               for b, g in ((state.index(i), goal_state.index(i)) for i in range(1,
board_len)))


def backtrace():

    current_node = goal_node

    while initial_state != current_node.state:

        if current_node.move == 1:
            movement = 'Up'
        elif current_node.move == 2:
            movement = 'Down'
        elif current_node.move == 3:
            movement = 'Left'
        else:
            movement = 'Right'
```

```python
        moves.insert(0, movement)
        current_node = current_node.parent

    return moves


def export(frontier, time):

    global moves

    moves = backtrace()
    file = open('output.txt', 'w')
    file.write("path_to_goal: " + str(moves))
    file.write("\ncost_of_path: " + str(len(moves)))
    file.write("\nnodes_expanded: " + str(nodes_expanded))
    file.write("\nfringe_size: " + str(len(frontier)))
    file.write("\nsearch_depth: " + str(goal_node.depth))
    file.write("\nrunning_time: " + format(time, '.8f'))
    file.close()


def read(configuration):

    global board_len, board_side

    data = configuration.split(",")

    for element in data:
        initial_state.append(int(element))

    board_len = len(initial_state)

    board_side = int(board_len ** 0.5)


def main():

    parser = argparse.ArgumentParser()

    parser.add_argument('algorithm')
    parser.add_argument('board')
    args = parser.parse_args()
```

```
    read(args.board)
    function = function_map[args.algorithm]

    start = timeit.default_timer()

    frontier = function(initial_state)

    stop = timeit.default_timer()

    export(frontier, stop-start)


function_map = {
    'bfs': bfs,
    'dfs': dfs,
    'ast': ast,
}

if __name__ == '__main__':
    main()
```

Output:

Initial State - 8,1,7,6,5,4,3,2,0

Bfs Result

```
path_to_goal: ['Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Down', 'Right', 'Up', 'Up',
cost_of_path: 22
nodes_expanded: 89958
fringe_size: 21587
search_depth: 22
running_time: 0.71301513
```

Dfs Result

```
path_to_goal: ['Up', 'Up', 'Left', 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Do
cost_of_path: 3688
nodes_expanded: 3784
fringe_size: 2946
search_depth: 3688
running_time: 0.04141871
```

A Star Result

```
path_to_goal: ['Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Down', 'Right', 'Up', 'Up',
cost_of_path: 22
nodes_expanded: 411
fringe_size: 246
search_depth: 22
running_time: 0.01191004
```

Conclusion

In This experiment, we tried to solve the 8. Puzzle problem using dfs and bfs. Along with that we also solved it using the A-Star algorithm to give a benchmark of comparison.

DFS was implemented using a stack, BFS was implemented using a queue, whereas A star algorithm was implemented using a heap(to find the lowest cost node quickly).

We observed that dfs was quick to solve the problem but it did not give the most optimum path i.e the shortest path to reach the required state. BFS though slower than bfs, gave the shortest path to reach the required state. In Addition, BFS had to expand more number of nodes compared to dfs which added to the time difference. Therefore to judge which algorithm is better, we believe that there are two fronts of comparison -

1. Time taken to solve - dfs is better than bfs
2. Finding the shortest/smallest number of moves - bfs is better than dfs

In addition, we tried a heuristic approach with the A-star algorithm which worked on the Manhattan distance between the required state and goal state. This algorithm gave substantially low running time along with the lower number of moves to check and expanded the lower number of nodes. This gives the best of both worlds, i.e less time taken to solve and finding the lowest number of moves to solve the problem. Thus this gives us a good benchmark to understand the limitations as well as optimizations required.