CSL 412 - Artificial Intelligence

Name of the Student: Kaustubh Shivshankar Shejole
Enrollment Number : BT20CSE112
ID Number : 25702
Department: Computer Science and Engineering Department, VNIT Nagpur

Assignment I

Use Iterative Deepening Search to solve the 8-tile puzzle.

Programmatically, randomly generate an initial state.
Check for its solvability.
If not solvable, randomly generate another initial state.
If solvable, use Iterative Deepening Search to get the sequence of moves for the solution.

Firstly I would like to present the algorithm of checking solvability that is used:

```
import random
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]


def count_inversions(state):
    inversions = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] > state[j] and state[i] != 0 and state[j] != 0:
                inversions += 1
    return inversions


def is_solvable(state, goal_state):
    inversions_initial_state = count_inversions(state)
    inversions_goal_state = count_inversions(goal_state)

    return (inversions_initial_state % 2 == inversions_goal_state % 2)


def generate_random_state():
    state = list(range(9))
    random.shuffle(state)
    return state


def find_solvable_initial_state(goal_state):
    while True:
```

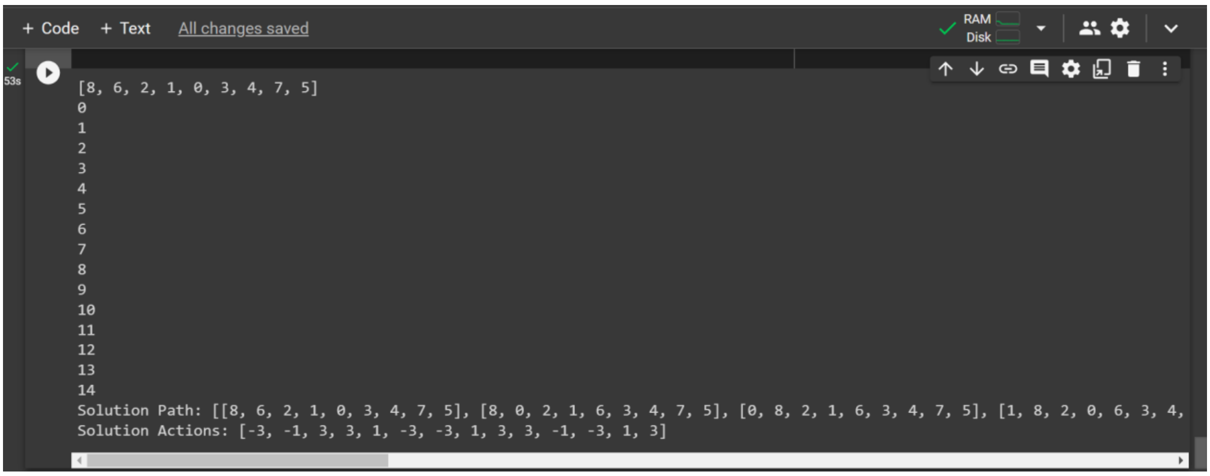
```
        initial_state = generate_random_state()
        if is_solvable(initial_state, goal_state) and initial_state != goal_state:
            return initial_state
```

The algorithm is based on inversions theory:

1. We can reach a goal state if the parity of inversions i.e., number_of_inversions modulo 2 , in initial state is equal to the parity of inversions in the goal state.
2. Our goal state is predefined as [1,2,3,4,5,6,7,8,0] where 0 denotes the blank tile.
3. An inversion is said to happen when a tile with a bigger number precedes a tile with a smaller number (except 0 as it is not a tile).
4. We can clearly see that the goal state has 0 inversions (all are in ascending order), so it has even parity.
5. So we require an initial state with even parity or even number of inversions.
6. Complexity of Checking Solvability is $\theta(n^2)$ which is a polynomial-time complexity and so it is allowable.
7. Working of the algorithm
    a. `generate_random_state ()`: generates any random state.
    b. `is_solvable(state, goal_state)`: checks the solvability by comparing inversions parity. We may take out the inversions_goal_state variable and make it as a global for more efficiency.
    c. `count_inversions(state)`: counts the number of inversions in a state.
    d. `find_solvable_initial_state(goal_state)`: runs a while loop till we get a solvable_initial_state different from the goal_state itself.
8. In this way the algorithm for checking solvability and returning a solvable initial state works.

For implementing the 8-tile puzzle using Iterative Deepening Depth-First Search , I used 3 Approaches. The approaches are as follows:

1. In the first approach, I used the pure iterative deepening depth-first search without considering any optimizations. For a goal node at depth 14 (considering the goal node at the depth 0)



Code:

```
'''
```

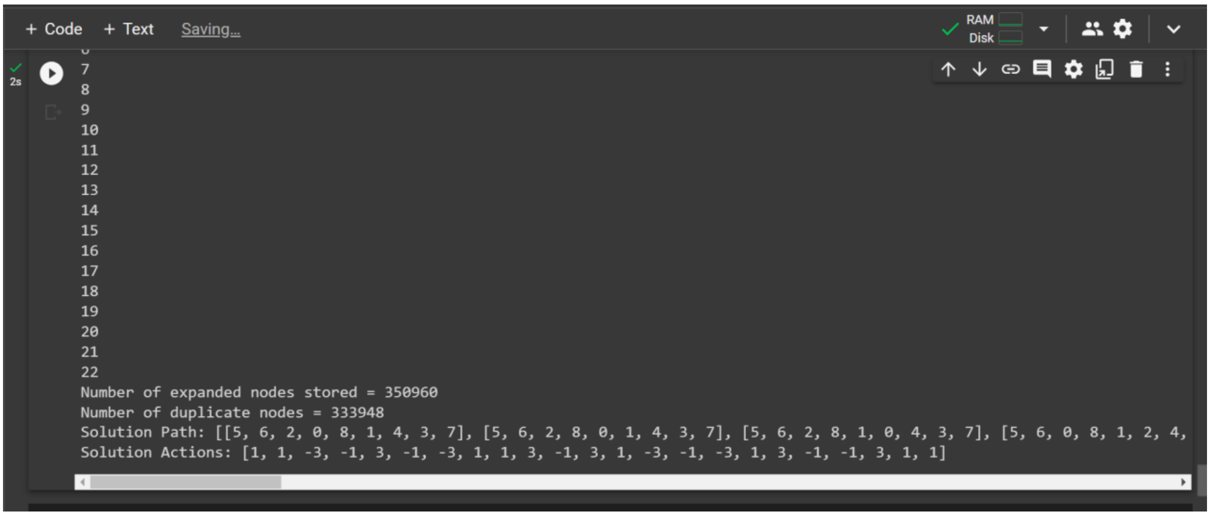
```
Author: BT20CSE112 Kaustubh Shivshankar Shejole
Description: The below code is the implementation of Iterative Deepening Depth
First Search for 8_tile problem
              without considering any optimization.
'''
import random
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]


def count_inversions(state):
    inversions = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] > state[j] and state[i] != 0 and state[j] != 0:
                inversions += 1
    return inversions


def is_solvable(state, goal_state):
    inversions_initial_state = count_inversions(state)
    inversions_goal_state = count_inversions(goal_state)

    return (inversions_initial_state % 2 == inversions_goal_state % 2)


def generate_random_state():
    state = list(range(9))
    random.shuffle(state)
    return state


def find_solvable_initial_state(goal_state):
    while True:
        initial_state = generate_random_state()
        if is_solvable(initial_state, goal_state) and initial_state != goal_state:
            return initial_state

def is_goal_state(state):
    return state == [1, 2, 3, 4, 5, 6, 7, 8, 0]

def apply_action(state, action):
    new_state = list(state)
    blank_index = new_state.index(0)
    new_index = blank_index + action
    new_state[blank_index], new_state[new_index] = new_state[new_index],
new_state[blank_index]
    return new_state

def get_actions(state):
```

```
    actions = []
    blank_index = state.index(0)
    if blank_index - 3 >= 0:  # Up
        actions.append(-3)
    if blank_index + 3 < 9:  # Down
        actions.append(3)
    if blank_index % 3 > 0:  # Left
        actions.append(-1)
    if blank_index % 3 < 2:  # Right
        actions.append(1)
    return actions




def depth_limited_dfs(state, depth_limit):
    if is_goal_state(state):
        return [state], []

    if depth_limit == 0:
        return None, []

    for action in get_actions(state):
        child_state = apply_action(state, action)
        if is_goal_state(child_state):
            return [state, child_state], [action]
        if(is_solvable(child_state,goal_state)):
          result, actions = depth_limited_dfs(child_state, depth_limit - 1)
          if result is not None:
              return [state] + result, [action] + actions
        else:
          print("False")
    return None, []

def iddfs_without_optimization(initial_state):
    depth_limit = 0

    while True:
        solution_path, actions = depth_limited_dfs(initial_state, depth_limit)
        print(depth_limit)
        if solution_path:
            return solution_path, actions

        depth_limit += 1

initial_state = find_solvable_initial_state(goal_state)
print(initial_state)

# Run IDDFS algorithm and print the solution path and actions
```

```python
solution_path, actions = iddfs_without_optimization(initial_state)
if solution_path:
    print("Solution Path:", solution_path)
    print("Solution Actions:", actions)
else:
    print("No solution found.") # not going to be executed !!!
```

2. In the second approach , I used an optimization:
   The optimization is as follows:
   When the states are expanded we get to know that the same states with the same depth are expanded again which is not optimal , so I created expanded_list with (tuple(state),depth_limit - 1) as an entity in expanded_list. It avoided about 94% of the nodes in expanded_list to be expanded so it speeded up the execution.

   Sample output image:



We can see about 3,33,948 nodes we encountered from 3,50,960 nodes in expanded_list.
Total nodes to be expanded in Iterative deepening depth-first search without optimization are equal to 333948 + 350960 = 684908.
Total nodes stored in Iterative deepening depth-first search with optimization are equal to 350960 nodes.

Code:

```
import random
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]


def count_inversions(state):
    inversions = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] > state[j] and state[i] != 0 and state[j] != 0:
                inversions += 1
    return inversions


def is_solvable(state, goal_state):
    inversions_initial_state = count_inversions(state)
    inversions_goal_state = count_inversions(goal_state)
```

```python
    return (inversions_initial_state % 2 == inversions_goal_state % 2)


def generate_random_state():
    state = list(range(9))
    random.shuffle(state)
    return state


def find_solvable_initial_state(goal_state):
    while True:
        initial_state = generate_random_state()
        if is_solvable(initial_state, goal_state) and initial_state != goal_state:
            return initial_state


def is_goal_state(state):
    return state == [1, 2, 3, 4, 5, 6, 7, 8, 0]


def apply_action(state, action):
    new_state = list(state)
    blank_index = new_state.index(0)
    new_index = blank_index + action
    new_state[blank_index], new_state[new_index] = new_state[new_index], new_state[blank_index]
    return new_state


def get_actions(state):
    actions = []
    blank_index = state.index(0)
    if blank_index - 3 >= 0:  # Up
        actions.append(-3)
    if blank_index + 3 < 9:  # Down
        actions.append(3)
    if blank_index % 3 > 0:  # Left
        actions.append(-1)
    if blank_index % 3 < 2:  # Right
        actions.append(1)
    return actions


def iddfs_optimized(initial_state):
    depth_limit = 0
    expanded_nodes = set()
    number_of_duplicates = 0
    while True:
        solution_path, actions, number_of_duplicates = depth_limited_dfs_optimized(
            initial_state, depth_limit, expanded_nodes, number_of_duplicates)
```

```
        if solution_path:
            print("Number of expanded nodes stored = "+str(len(expanded_nodes)))
            print("Number of duplicate nodes = " + str(number_of_duplicates))
            return solution_path, actions
        print(depth_limit)
        depth_limit += 1


def depth_limited_dfs_optimized(state, depth_limit, expanded_nodes, number_of_duplicates):
    if is_goal_state(state):
        return [state], [], number_of_duplicates

    if depth_limit == 0:
        return None, [], number_of_duplicates

    expanded_nodes.add((tuple(state), depth_limit))

    for action in get_actions(state):
        child_state = apply_action(state, action)
        if is_goal_state(child_state):
            return [state, child_state], [action], number_of_duplicates
        if (tuple(child_state), depth_limit - 1) not in expanded_nodes:
            result, actions, number_of_duplicates = depth_limited_dfs_optimized(
                child_state, depth_limit - 1, expanded_nodes, number_of_duplicates)
            if result is not None:
                return [state] + result, [action] + actions, number_of_duplicates
        else:
            number_of_duplicates = number_of_duplicates + 1

    return None, [], number_of_duplicates


# Example initial state
initial_state = find_solvable_initial_state(goal_state)
print("Initial state = "+str(initial_state))

# Run IDDFS algorithm and print the solution path and actions
solution_path, actions = iddfs_optimized(initial_state)
if solution_path:
    print("Solution Path:", solution_path)
    print("Solution Actions:", actions)
else:
    print("No solution found.")
```

Advantages of this approach:

1. It is time-efficient.
2. It uses optimization to not allow duplicate node with the depth_remaining_to_be_explored as same i.e., depth_limit -1 to be expanded.

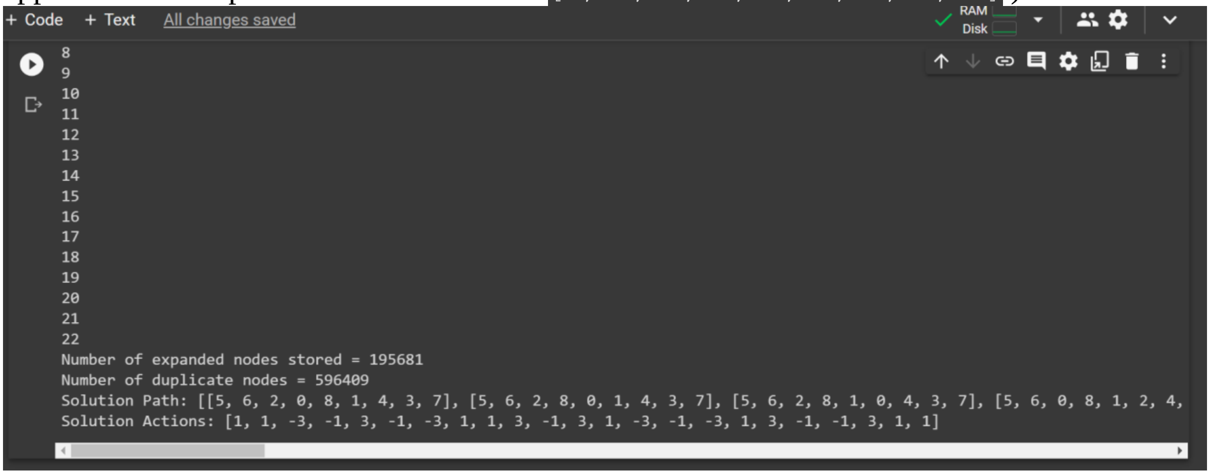
3. Due to the above expanded list , algorithm becomes time efficient.

Disadvantages of this approach:

1. As we can see here the expanded_list stores the entities of (state,depth_limit -1) for every iteration so it becomes larger and we need more space to make it time efficient.

3. My third approach was to reduce the size of expanded_list so that we will be little space-efficient and a little time-inefficient. Here in this approach , I did a minor change i.e., for each depth emptying the expanded_list so now it will take a little more time (4 seconds or 5 seconds) but a little less space (for example instead of storing 350960 entities, it will store at max 195681 entities , about 44% reduction in space).

Output Image: (I copied the initial state of 2nd approach and tried to get results for this approach for comparison. Initial state was `[5, 6, 2, 0, 8, 1, 4, 3, 7]`.)



```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
Number of expanded nodes stored = 195681
Number of duplicate nodes = 596409
Solution Path: [[5, 6, 2, 0, 8, 1, 4, 3, 7], [5, 6, 2, 8, 0, 1, 4, 3, 7], [5, 6, 2, 8, 1, 0, 4, 3, 7], [5, 6, 0, 8, 1, 2, 4,
Solution Actions: [1, 1, -3, -1, 3, -1, -3, 1, 1, 3, -1, 3, 1, -3, -1, -3, 1, 3, -1, -1, 3, 1, 1]
```

Analysis:

Here the number of duplicate nodes checked are more, it does not imply that this is an optimized algorithm than the 2nd approach one it is just because there would be more expansions here so more duplicate nodes would be checked. For the last depth it would be 194898 nodes. It also avoided about 94% nodes of the size of expanded list in each iteration not to be expanded.

Code:

```
import random
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]


def count_inversions(state):
    inversions = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] > state[j] and state[i] != 0 and state[j] != 0:
                inversions += 1
    return inversions


def is_solvable(state, goal_state):
    inversions_initial_state = count_inversions(state)
    inversions_goal_state = count_inversions(goal_state)

    return (inversions_initial_state % 2 == inversions_goal_state % 2)
```

```python
def generate_random_state():
    state = list(range(9))
    random.shuffle(state)
    return state


def find_solvable_initial_state(goal_state):
    while True:
        initial_state = generate_random_state()
        if is_solvable(initial_state, goal_state) and initial_state != goal_state:
            return initial_state

def is_goal_state(state):
    return state == [1, 2, 3, 4, 5, 6, 7, 8, 0]


def apply_action(state, action):
    new_state = list(state)
    blank_index = new_state.index(0)
    new_index = blank_index + action
    new_state[blank_index], new_state[new_index] = new_state[new_index],
new_state[blank_index]
    return new_state


def get_actions(state):
    actions = []
    blank_index = state.index(0)
    if blank_index - 3 >= 0:  # Up
        actions.append(-3)
    if blank_index + 3 < 9:  # Down
        actions.append(3)
    if blank_index % 3 > 0:  # Left
        actions.append(-1)
    if blank_index % 3 < 2:  # Right
        actions.append(1)
    return actions


def iddfs_optimized(initial_state):
    depth_limit = 0

    number_of_duplicates = 0
    while True:
        expanded_nodes = set()
        solution_path, actions, number_of_duplicates = depth_limited_dfs_optimized(
            initial_state, depth_limit, expanded_nodes, number_of_duplicates)
        if solution_path:
            print("Number of expanded nodes stored = "+str(len(expanded_nodes)))
```

```python
            print("Number of duplicate nodes = " + str(number_of_duplicates))
            return solution_path, actions
        print(depth_limit)
        depth_limit += 1


def depth_limited_dfs_optimized(state, depth_limit, expanded_nodes, number_of_duplicates):
    if is_goal_state(state):
        return [state], [], number_of_duplicates

    if depth_limit == 0:
        return None, [], number_of_duplicates

    expanded_nodes.add((tuple(state), depth_limit))

    for action in get_actions(state):
        child_state = apply_action(state, action)
        if is_goal_state(child_state):
            return [state, child_state], [action], number_of_duplicates
        if (tuple(child_state), depth_limit - 1) not in expanded_nodes:
            result, actions, number_of_duplicates = depth_limited_dfs_optimized(
                child_state, depth_limit - 1, expanded_nodes, number_of_duplicates)
            if result is not None:
                return [state] + result, [action] + actions, number_of_duplicates
        else:
            number_of_duplicates = number_of_duplicates + 1

    return None, [], number_of_duplicates


# Example initial state
# initial_state = find_solvable_initial_state(goal_state)
initial_state = [5, 6, 2, 0, 8, 1, 4, 3, 7]
print("Initial state = "+str(initial_state))

# Run IDDFS algorithm and print the solution path and actions
solution_path, actions = iddfs_optimized(initial_state)
if solution_path:
    print("Solution Path:", solution_path)
    print("Solution Actions:", actions)
else:
    print("No solution found.")
```

Advantages:

1. Take less space for expanded nodes from the 2nd approach.

Disadvantages:

1. The time complexity is more than the 2nd approach due to redundant expansions as the data of expanded entities at each iteration becomes null.

Thank You Sir