AIML LAB 5

U18CO021: SAHIL BONDRE

Implement 8 Puzzle problem using below algorithms in prolog.

Compare the complexity of both algorithms.

Which algorithm is best suited for implementing 8 Puzzle problem and why ?

1. Breadth First Search

2. Depth First Search

3. Uniform Cost Search

4. Greedy Best First Search

Comparison of Algorithms

Algorithm	Complete	Optimal	Time	Space
BFS	Yes	Yes	O(b^d)	O(b^d)
DFS	Yes	No	O(b^d)	O(bd)
UCS	Yes	Yes	O(b^[C/e])	O(b^[C/e])
GBFS	No	No	O(b^m)	O(b^m)

Note:

m is maximum depth of search space

C is cost of optimal solution

e is step cost

Code

Board class is used to define the operations on the board like moving up or down.

__hash__ and __lt__ are defined for storing in set and heaps respectively.

```
class Board:
    def __init__(self, state=[1, 2, 3, 8, 0, 4, 7, 6, 5]):
        # 0 implies empty state
        assert len(state) == 9
        self.value = state

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

    def __hash__(self):
        return "".join(str(x) for x in self.value).__hash__()

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

    def near_completion(self):
        complete = [1, 2, 3, 8, 0, 4, 7, 6, 5]
        completion = 9
        for i in range(9):
            if self.value[i] == complete[i]:
                completion -= 1
        return completion

    def _find_space(self):
        for i in range(9):
            if self.value[i] == 0:
                return i

    def is_left(self):
        location = self._find_space()
        if location in [0, 3, 6]:
            return False
        return True

    def is_right(self):
        location = self._find_space()
        if location in [2, 5, 8]:
            return False
        return True
```

```
    def is_up(self):
        location = self._find_space()
        if location in [0, 1, 2]:
            return False
        return True

    def is_down(self):
        location = self._find_space()
        if location in [6, 7, 8]:
            return False
        return True

    def up(self):
        state = list.copy(self.value)
        if not self.is_up():
            raise Exception(f"Up not possible, state: {self.value}")
        i = self._find_space()
        state[i], state[i - 3] = state[i - 3], state[i]
        return Board(state)

    def down(self):
        state = list.copy(self.value)
        if not self.is_down():
            raise Exception(f"Down not possible, state: {self.value}")
        i = self._find_space()
        state[i], state[i + 3] = state[i + 3], state[i]
        return Board(state)

    def left(self):
        state = list.copy(self.value)
        if not self.is_left():
            raise Exception(f"Left not possible, state: {self.value}")
        i = self._find_space()
        state[i], state[i - 1] = state[i - 1], state[i]
        return Board(state)

    def right(self):
        state = list.copy(self.value)
        if not self.is_right():
            raise Exception(f"Right not possible, state: {self.value}")
        i = self._find_space()
        state[i], state[i + 1] = state[i + 1], state[i]
        return Board(state)

    def expand(self):
```

```
        res = {}
        if self.is_left():
            res[self.left()] = "l"
        if self.is_right():
            res[self.right()] = "r"
        if self.is_up():
            res[self.up()] = "u"
        if self.is_down():
            res[self.down()] = "d"

        return res

    def print(self):
        print(f"{self.value[0]} | {self.value[1]} | {self.value[2]}")
        print(f"-   -   -")
        print(f"{self.value[3]} | {self.value[4]} | {self.value[5]}")
        print(f"-   -   -")
        print(f"{self.value[6]} | {self.value[7]} | {self.value[8]}")
        print()
```

CostPath is a utility class used for Uniform Cost Search

```
class CostPath:
    def __init__(self, board, cost):
        self.board = board
        self.cost = cost

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

Index.py contains the implementation of all 4 algorithms

```
from board import Board
from collections import deque
import heapq
from typing import Deque, Dict, List
from cost_path import CostPath


def bfs(start):
    # start = Board([2, 8, 1, 0, 4, 3, 7, 6, 5])
```

```
    # start = Board([1, 2, 3, 0, 8, 4, 7, 6, 5])
    goal = Board()

    q: Deque[Board] = deque()
    m: Dict[Board, List[str]] = {start: []}
    q.append(start)

    while q:
        node = q.popleft()
        path = m[node]

        if node == goal:
            print("Found!")
            print(path)
            return

        neighbours = node.expand()

        for neighbour in neighbours:
            if neighbour not in m:
                q.append(neighbour)
                newPath = list.copy(path)
                newPath.append(neighbours[neighbour])
                m[neighbour] = newPath


def dfs(start):
    # start = Board([2, 8, 1, 0, 4, 3, 7, 6, 5])
    goal = Board()

    q: Deque[Board] = deque()
    m: Dict[Board, List[str]] = {start: []}
    q.append(start)

    while q:
        node = q.pop()
        path = m[node]

        if node == goal:
            print("Found!")
            print(path)
            return

        neighbours = node.expand()

        for neighbour in neighbours:
```

```
            if neighbour not in m:
                q.append(neighbour)
                newPath = list.copy(path)
                newPath.append(neighbours[neighbour])
                m[neighbour] = newPath


def gbfs(start):
    goal = Board()

    q = []
    m: Dict[Board, List[str]] = {start: []}
    heapq.heappush(q, start)

    while q:
        node = heapq.heappop(q)
        path = m[node]

        if node == goal:
            print("Found!")
            print(path)
            return

        neighbours = node.expand()

        for neighbour in neighbours:
            if neighbour not in m:
                heapq.heappush(q, neighbour)
                newPath = list.copy(path)
                newPath.append(neighbours[neighbour])
                m[neighbour] = newPath


def ucs(start):
    goal = Board()

    q = []
    m: Dict[Board, List[str]] = {start: []}
    heapq.heappush(q, CostPath(start, 0))

    while q:
        cp = heapq.heappop(q)
        node = cp.board
        cost = cp.cost
        path = m[node]
```

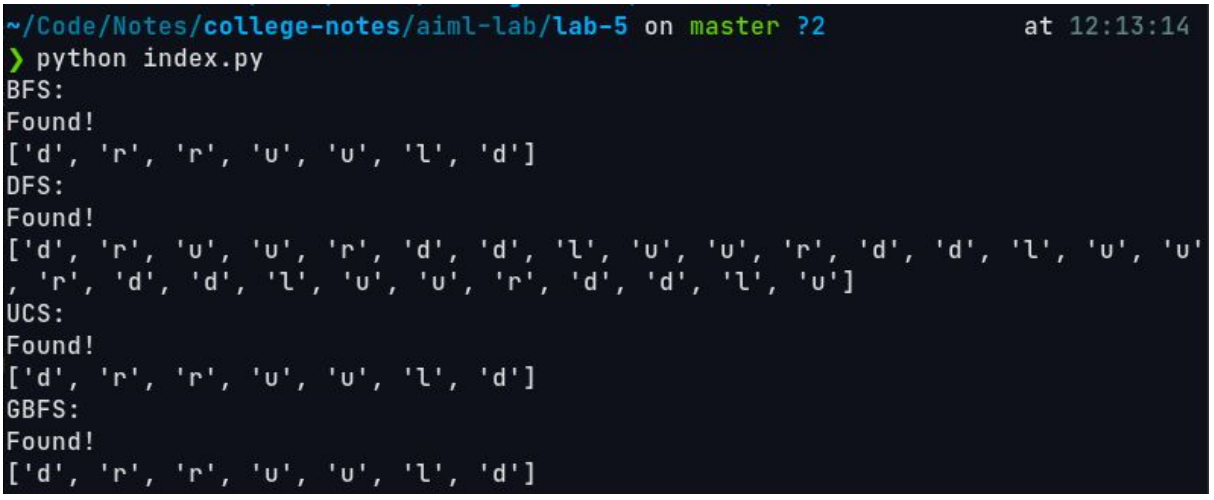
```
        if node == goal:
            print("Found!")
            print(path)
            return

        neighbours = node.expand()

        for neighbour in neighbours:
            if neighbour not in m:
                heapq.heappush(q, CostPath(neighbour, cost + 1))
                newPath = list.copy(path)
                newPath.append(neighbours[neighbour])
                m[neighbour] = newPath


start = Board().up().right().down().down().left().left().up()

bfs(start)
dfs(start)
ucs(start)
gbfs(start)
```



Note: DFS is not optimal hence finds a longer solution while the rest algorithms find the same optimal solution.

Uniform Cost Search is best suited for solving this problem as it takes least time and space but is Complete and Optimal too.