Name: Mihir Thakkar
Class: TY A
Roll No: 59
Srn: 201901267

AI ASSIGNMENT - 4
A* ALGORITHM:
1. ROBOT NAVIGATION: CODE:

```
EMPTY = 0
WALL = 1
SELECTED = 2

class Node:
    def __init__(self, x, y, prev = None):
        self.x = x
        self.y = y
        self.prev = prev

    def __eq__(self, other):
        if isinstance(other, Node):
            return self.x == other.x and self.y == other.y
        elif isinstance(other, tuple):
            return self.x == other[0] and self.y == other[1]

    def __repr__(self):
        return str((self.x, self.y))

    def __hash__(self):
        return (self.x, self.y).__hash__()

    def man_dist(self, to) -> int:
        return abs(self.x - to.x) + abs(self.y - to.y)

    def heu(self, goal) -> int:
        return self.man_dist(goal)

    def neighbours(self, board) -> list:
        neigh = [
            (self.x, self.y - 1),
            (self.x - 1, self.y),
            (self.x + 1, self.y),
            (self.x, self.y + 1),
        ]
```

```
        ugly_neighbours = []
        for n in neigh:
            x, y = n

            if x < 0 or y < 0:
                ugly_neighbours.append((x, y))
            elif x >= len(board) or y >= len(board[0]):
                ugly_neighbours.append((x, y))
            elif board[x][y] == WALL:
                ugly_neighbours.append((x, y))


        for ugly in ugly_neighbours:
            neigh.remove(ugly)

        return [Node(n[0], n[1], self) for n in neigh]

Board = list[list[Node]]

def display_board(board, initial: Node, goal: Node) -> None:
    print('', '-' * len(board[0] * 3))

    for i, row in enumerate(board):
        print(end='|')
        for j, col in enumerate(row):
            if Node(i, j) == initial:
                ch='S'
            elif Node(i, j) == goal:
                ch='G'
            elif col == EMPTY:
                ch=' '
            elif col == WALL:
                ch='*'
            elif col == SELECTED:
                ch='$'
            print("{:>3}".format(ch), end='')
        print(end='|\n')

    print('', '-' * len(board[0]) * 3)

def start(board, initial: Node, goal: Node) -> Node:
    opened = []
    closed = [Node(initial.x, initial.y)]
```

```
    d_board = []
    for row in board:
        d_board.append([])
        for col in row:
            d_board[-1].append(col)

    previous = set()
    while len(closed) > 0:
        closed.sort(key = lambda n: n.heu(initial) + n.heu(goal))
        current = closed.pop(0)
        opened.append(current)

        if current in previous:
            continue
        else:
            previous.add(current)

        d_board[current.x][current.y] = SELECTED
        print("\n\nOpen List : ", opened)
        print("\nClose List : ", closed)
        display_board(d_board, initial, goal)

        if current == goal:
            print("\n\n\nReached goal!")
            break

        for neighbour in current.neighbours(board):
            if neighbour not in previous:
                closed.append(neighbour)

    return current

def safety_check(board, initial: tuple[int, int], goal: tuple[int, int]):
    if board[initial[0]][initial[1]] == WALL:
        raise ValueError("\nInitial Position cannot be a wall")
    if board[goal[0]][goal[1]] == WALL:
        raise ValueError("\nGoal Position cannot be a wall")


def main():
    board = [
        [0, 0, 0, 0, 0, 0, 0, 0, 0,],
        [0, 1, 0, 0, 1, 1, 1, 1, 0,],
        [0, 1, 0, 0, 0, 0, 0, 1, 0,],
```

```
        [0, 0, 1, 1, 1, 1, 1, 1, 0,],
        [0, 0, 0, 0, 0, 0, 0, 0, 0,],
    ]

    initial = (3, 0)
    goal = (2, 5)
    safety_check(board, initial, goal)

    end = start(board, Node(initial[0], initial[1]), Node(goal[0], goal[1]))
    print("\n\n\nBest route : ")
    path = []
    while end:
        path.append((end.x, end.y))
        end = end.prev
    d_board = []
    for row in board:
        d_board.append([])
        for col in row:
            d_board[-1].append(col)
    for (x, y) in path:
        d_board[x][y] = SELECTED
    display_board(d_board, initial, goal)


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

OUTPUT:

```
Open List :  [(3, 0)]

Close List :  []
 -------------------------
|                          |
|     *        *  *  *  *  |
|     *           G     *  |
|  S     *  *  *  *  *  *  |
|                          |
 -------------------------


Open List :  [(3, 0), (2, 0)]

Close List :  [(3, 1), (4, 0)]
 -------------------------
|                          |
|     *        *  *  *  *  |
|  $  *           G     *  |
|  S     *  *  *  *  *  *  |
|                          |
 -------------------------


Open List :  [(3, 0), (2, 0), (3, 1)]

Close List :  [(4, 0), (1, 0)]
 -------------------------
|                          |
|     *        *  *  *  *  |
|  $  *           G     *  |
|  S  $  *  *  *  *  *  *  |
|                          |
 -------------------------


Open List :  [(3, 0), (2, 0), (3, 1), (4, 0)]

Close List :  [(1, 0), (4, 1)]
 -------------------------
|                          |
|     *        *  *  *  *  |
|  $  *           G     *  |
|  S  $  *  *  *  *  *  *  |
|  $                       |
```

```
Open List :  [(3, 0), (2, 0), (3, 1), (4, 0), (1, 0), (4, 1), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (0, 0), (4, 6), (0, 1), (0, 2), (1, 2), (2, 2)]

Close List :  [(1, 3), (0, 3), (4, 7)]
 -------------------------
|  $  $  $                 |
|  $  *  $     *  *  *  *  |
|  $  *  $        G     *  |
|  S  $  *  *  *  *  *  *  |
|  $  $  $  $  $  $  $     |
 -------------------------


Open List :  [(3, 0), (2, 0), (3, 1), (4, 0), (1, 0), (4, 1), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (0, 0), (4, 6), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3)]

Close List :  [(1, 3), (0, 3), (4, 7)]
 -------------------------
|  $  $  $                 |
|  $  *  $     *  *  *  *  |
|  $  *  $  $     G     *  |

Close List :  [(1, 3), (1, 3), (0, 3), (4, 7)]
 -------------------------
|  $  $  $                 |
|  $  *  $     *  *  *  *  |
|  $  *  $  $  $  G     *  |
|  S  $  *  *  *  *  *  *  |
|  $  $  $  $  $  $  $     |
 -------------------------



Reached goal!


Best route :
 -------------------------
|  $  $  $                 |
|  $  *  $     *  *  *  *  |
|  $  *  $  $  $  G     *  |
|  S     *  *  *  *  *  *  |
|                          |
 -------------------------
```

1. CITY PROBLEM:

CODE:

```
SOURCE = "Pune"
DESTINATION = "Aurangabad"

DISTANCE_FROM_DESTINATION = {
    "Solapur": 340,
    "Satara": 177,
    "Navi Mumbai": 181,
    "Nashik": 133,
    "Ahmednagar": 180,
    "Aurangabad": 0,
    "Nanded": 451,
    "Latur": 373,
    "Pune": 424,
}

GRAPH = {}

def put_city(city1: str, city2: str, weight: int) -> None:
    global GRAPH

    if city1 in GRAPH:
        GRAPH[city1].append((city2, weight))
    else:
        GRAPH[city1] = [(city2, weight)]

    if city2 in GRAPH:
        GRAPH[city2].append((city1, weight))
    else:
        GRAPH[city2] = [(city1, weight)]

def generate_map() -> None:
    put_city("Pune", "Navi Mumbai", 106)
    put_city("Pune", "Satara", 112)
    put_city("Pune", "Solapur", 234)
    put_city("Solapur", "Satara", 203)
    put_city("Solapur", "Latur", 104)
    put_city("Nanded", "Latur", 113)
    put_city("Nanded", "Aurangabad", 221)
    put_city("Nashik", "Aurangabad", 159)
```

```
    put_city("Nanded", "Ahmednagar", 267)
    put_city("Pune", "Ahmednagar", 120)
    put_city("Nashik", "Pune", 165)
    put_city("Nashik", "Navi Mumbai", 136)


def distance(from_: str, to: str) -> int:
    if from_ == to:
        return 0

    array = GRAPH[from_]
    for name, weight in array:
        if name == to:
            return weight

    return 10 ** 10

class Node:
    def __init__(self, city: str, prev: object, weight: int) -> None:
        self.city = city
        self.prev = prev
        self.weight = weight

    def __repr__(self) -> str:
        return self.city

    def __eq__(self, other) -> bool:
        if isinstance(other, Node):
            return self.city == other.city
        elif isinstance(other, str):
            return self.city == other
        else:
            return False

def objective(city, parent) -> int:
    return distance(city, parent) + DISTANCE_FROM_DESTINATION[city]

def main() -> None:
    generate_map()
    print("\nStart city : Pune")
    print("\nDestination city : Aurangabad")

    current = Node(SOURCE, None, 0)
    opened = []
```

```
    closed = [current]

    while len(closed) > 0:
        print("\nOpened list : ", opened)
        print("\nClosed list : ", closed)

        closed.sort(
                key=lambda node: objective(node.city, current),
                reverse=True
        )

        current = closed.pop()

        if current not in opened:
            opened.append(current)

        if current == DESTINATION:
            print("\n\n\nReached Destination!!".center(40, ' '))
            break

        print("\nSelected City : ", current)
        print(f"\nThe neighbouring cities of {current} are : ")
        for name, dist in GRAPH[current.city]:
            print(f"{name:>12} with distance {dist}")

            if name not in opened:
                closed.append(Node(name, current, dist))
        print()

    path = []
    while current:
        path.append(current)
        current = current.prev

    print("\n\nComplete Path : ")
    for city in path[::-1]:
        if city.prev:
            print(f"{city.prev.city} to {city.city} with distance {city.weight}")

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

OUTPUT:

```
Start city : Pune

Destination city : Aurangabad

Opened list :  []

Closed list :  [Pune]

Selected City :  Pune

The neighbouring cities of Pune are :
 Navi Mumbai with distance 106
      Satara with distance 112
     Solapur with distance 234
  Ahmednagar with distance 120
      Nashik with distance 165


Opened list :  [Pune]

Closed list :  [Navi Mumbai, Satara, Solapur, Ahmednagar, Nashik]

Selected City :  Navi Mumbai

The neighbouring cities of Navi Mumbai are :
        Pune with distance 106
      Nashik with distance 136


Opened list :  [Pune, Navi Mumbai]

Closed list :  [Solapur, Ahmednagar, Nashik, Satara, Nashik]

Selected City :  Nashik

The neighbouring cities of Nashik are :
  Aurangabad with distance 159
        Pune with distance 165
 Navi Mumbai with distance 136


Opened list :  [Pune, Navi Mumbai, Nashik]

Closed list :  [Solapur, Ahmednagar, Satara, Nashik, Aurangabad]
```

```
Selected City :  Navi Mumbai

The neighbouring cities of Navi Mumbai are :
        Pune with distance 106
      Nashik with distance 136


Opened list :  [Pune, Navi Mumbai]

Closed list :  [Solapur, Ahmednagar, Nashik, Satara, Nashik]

Selected City :  Nashik

The neighbouring cities of Nashik are :
  Aurangabad with distance 159
        Pune with distance 165
 Navi Mumbai with distance 136


Opened list :  [Pune, Navi Mumbai, Nashik]

Closed list :  [Solapur, Ahmednagar, Satara, Nashik, Aurangabad]

Selected City :  Nashik

The neighbouring cities of Nashik are :
  Aurangabad with distance 159
        Pune with distance 165
 Navi Mumbai with distance 136


Opened list :  [Pune, Navi Mumbai, Nashik]

Closed list :  [Solapur, Ahmednagar, Satara, Aurangabad, Aurangabad]


Reached Destination!!


Complete Path :
Pune to Nashik with distance 165
Nashik to Aurangabad with distance 159
```

2. 8 PUZZLE:

CODE:

```
GOAL = {
   1: 1, 2: 2, 3: 3,
   4: 8, 5: -1, 6: 4,
   7: 7, 8: 6, 9: 5,
}
t_board = dict[int, int]

class Node:
   def __init__(self, board: dict[int, int], steps: int, prev=None, op=None):
      self.board = board.copy()
      self.steps = steps
      self.prev = prev
      self.operation = op

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

   def __hash__(self) -> int:
      return heuristic(self.board)

   def __repr__(self):
      return str(self.heu())

   def heu(self) -> int:
      return self.steps + heuristic(self.board)


def get_empty_pos(puzzle: t_board) -> int:
   for key in puzzle:
      if puzzle[key] == -1:
         return key


def move_up(puzzle: t_board):
   pos = get_empty_pos(puzzle)
   puzzle[pos], puzzle[pos - 3] = puzzle[pos - 3], puzzle[pos]
   return puzzle
```

```
def move_down(puzzle: t_board):
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos + 3] = puzzle[pos + 3], puzzle[pos]
    return puzzle


def move_right(puzzle: t_board):
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos + 1] = puzzle[pos + 1], puzzle[pos]
    return puzzle


def move_left(puzzle: t_board):
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos - 1] = puzzle[pos - 1], puzzle[pos]
    return puzzle


def get_available_operations(puzzle: t_board):
    operations = {
        move_up,
        move_left, move_right,
        move_down,
    }
    empty_pos = get_empty_pos(puzzle)

    if empty_pos in {1, 2, 3}:
        operations.remove(move_up)
    if empty_pos in {7, 8, 9}:
        operations.remove(move_down)
    if empty_pos in {3, 6, 9}:
        operations.remove(move_right)
    if empty_pos in {1, 4, 7}:
        operations.remove(move_left)

    return operations

def get_operation_name(op):
    return {
        move_up: "\nMove Empty Slot UP",
        move_down: "\nMove Empty Slot DOWN",
        move_left: "\nMove Empty Slot LEFT",
        move_right: "\nMove Empty Slot RIGHT",
    }[op]
```

```
def heuristic(puzzle: t_board) -> int:
    score = 0

    for key in puzzle:
        if puzzle[key] != GOAL[key]:
            score += 1

    return score

def safe_get(puzzle: t_board, key: int) -> str:
    if puzzle[key] != -1:
        return puzzle[key]
    return '@'

def display_board(puzzle: dict[int, int]):
    print(
        str(safe_get(puzzle, 1)).center(3, ' ') +
        str(safe_get(puzzle, 2)).center(3, ' ') +
        str(safe_get(puzzle, 3)).center(3, ' ')
    )
    print('-' * 10)
    print(
        str(safe_get(puzzle, 4)).center(3, ' ') +
        str(safe_get(puzzle, 5)).center(3, ' ') +
        str(safe_get(puzzle, 6)).center(3, ' ')
    )
    print('-' * 10)
    print(
        str(safe_get(puzzle, 7)).center(3, ' ') +
        str(safe_get(puzzle, 8)).center(3, ' ') +
        str(safe_get(puzzle, 9)).center(3, ' ')
    )

def start(puzzle) -> Node:
    root = Node(puzzle, 0)
    opened = []
    closed = [root]
    steps = 0
    previous = set()
    best_score = 1000

    while len(closed) > 0:
        #print("\n\nOpened:", opened)
```

```
        #print("\nClosed:", closed)
        steps += 1
        closed.sort(key=lambda n: n.heu())
        current = closed.pop(0)
        opened.append(current)
        score = heuristic(current.board)

        if score == 0:
            return current

        if current in previous:
            continue
        else:
            previous.add(current)

        if score < best_score:
            best_score = score

        for operation in get_available_operations(current.board):
            child = Node(operation(current.board.copy()), steps, current, operation)
            closed.append(child)


    return current


def main():
    puzzle = {
        1: 1, 2: -1, 3: 3,
        4: 8, 5: 2, 6: 6,
        7: 7, 8: 5, 9: 4,
    }

    print("\nInitial State : ")
    display_board(puzzle)
    print("\nGoal State : ")
    display_board(GOAL)
    node = start(puzzle)


    path: list[Node] = []

    while node:
        path.append(node)
```

```
        node = node.prev
    path = path[::-1]
    path = path[1:]

    for i, node in enumerate(path):
        print(f"\n\nStep {i + 1} ")
        if node.operation:
            print(get_operation_name(node.operation))

        display_board(node.board)
        print(f"\nObjective Function Score : {heuristic(node.board) + i + 1}")

    if heuristic(path[-1].board) != 0:
        print("\nGoal State not Achieved!")
    else:
        print("\nGoal State Achieved")


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

OUTPUT:

```
Initial State :
 1  @  3
----------
 8  2  6
----------
 7  5  4

Goal State :
 1  2  3
----------
 8  @  4
----------
 7  6  5


Step 1

Move Empty Slot DOWN
 1  2  3
----------
 8  @  6
----------
 7  5  4

Objective Function Score : 4


Step 2

Move Empty Slot RIGHT
 1  2  3
----------
 8  6  @
----------
 7  5  4

Objective Function Score : 6
```

```
Step 3

Move Empty Slot DOWN
 1  2  3
----------
 8  6  4
----------
 7  5  @

Objective Function Score : 6


Step 4

Move Empty Slot LEFT
 1  2  3
----------
 8  6  4
----------
 7  @  5

Objective Function Score : 6


Step 5

Move Empty Slot UP
 1  2  3
----------
 8  @  4
----------
 7  6  5

Objective Function Score : 5

Goal State Achieved
```

BSF ALGORITHM:

3. ROBOT NAVIGATION:

CODE:

```
EMPTY = 0
WALL = 1
SELECTED = 2

class Node:
   def __init__(self, x, y, prev = None):
      self.x = x
      self.y = y
      self.prev = prev

   def __eq__(self, other):
```

```
        if isinstance(other, Node):
            return self.x == other.x and self.y == other.y
        elif isinstance(other, tuple):
            return self.x == other[0] and self.y == other[1]

    def __repr__(self):
        return str((self.x, self.y))

    def __hash__(self):
        return (self.x, self.y).__hash__()

    def man_dist(self, to) -> int:
        return abs(self.x - to.x) + abs(self.y - to.y)

    def heu(self, goal) -> int:
        return self.man_dist(goal)

    def neighbours(self, board) -> list:
        neigh = [
            (self.x, self.y - 1),
            (self.x - 1, self.y),
            (self.x + 1, self.y),
            (self.x, self.y + 1),
        ]

        ugly_neighbours = []
        for n in neigh:
            x, y = n

            if x < 0 or y < 0:
                ugly_neighbours.append((x, y))
            elif x >= len(board) or y >= len(board[0]):
                ugly_neighbours.append((x, y))
            elif board[x][y] == WALL:
                ugly_neighbours.append((x, y))


        for ugly in ugly_neighbours:
            neigh.remove(ugly)

        return [Node(n[0], n[1], self) for n in neigh]

Board = list[list[Node]]
```

```
def display_board(board, initial: Node, goal: Node) -> None:
    print('', '-' * len(board[0] * 3))

    for i, row in enumerate(board):
        print(end='|')
        for j, col in enumerate(row):
            if Node(i, j) == initial:
                ch='S'
            elif Node(i, j) == goal:
                ch='G'
            elif col == EMPTY:
                ch=' '
            elif col == WALL:
                ch='*'
            elif col == SELECTED:
                ch='$'
            print("{:>3}".format(ch), end='')
        print(end='|\n')

    print('', '-' * len(board[0]) * 3)

def start(board, initial: Node, goal: Node) -> Node:
    opened = []
    closed = [Node(initial.x, initial.y)]
    d_board = []
    for row in board:
        d_board.append([])
        for col in row:
            d_board[-1].append(col)

    previous = set()
    while len(closed) > 0:
        closed.sort(key = lambda n: n.heu(goal))
        current = closed.pop(0)
        opened.append(current)

        if current in previous:
            continue
        else:
            previous.add(current)

        d_board[current.x][current.y] = SELECTED
        print("\n\nOpen List : ", opened)
        print("\nClose List : ", closed)
```

```
        display_board(d_board, initial, goal)

        if current == goal:
            print("\n\n\nReached goal!!!")
            break

        for neighbour in current.neighbours(board):
            if neighbour not in previous:
                closed.append(neighbour)

    return current

def safety_check(board, initial: tuple[int, int], goal: tuple[int, int]):
    if board[initial[0]][initial[1]] == WALL:
        raise ValueError("\nInitial Position cannot be a wall")
    if board[goal[0]][goal[1]] == WALL:
        raise ValueError("\nGoal Position cannot be a wall")


def main():
    board = [
        [0, 0, 0, 0, 0, 0, 0, 0, 0,],
        [0, 1, 0, 0, 1, 1, 1, 1, 0,],
        [0, 1, 0, 0, 0, 0, 0, 1, 0,],
        [0, 0, 1, 1, 1, 1, 1, 1, 0,],
        [0, 0, 0, 0, 0, 0, 0, 0, 0,],
    ]

    initial = (3, 0)
    goal = (2, 5)
    safety_check(board, initial, goal)

    end = start(board, Node(initial[0], initial[1]), Node(goal[0], goal[1]))
    print("\nBest Route : ")
    path = []
    while end:
        path.append((end.x, end.y))
        end = end.prev
    d_board = []
    for row in board:
        d_board.append([])
        for col in row:
            d_board[-1].append(col)
    for (x, y) in path:
```

```
        d_board[x][y] = SELECTED
    display_board(d_board, initial, goal)


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

OUTPUT:

```
Open List :  [(3, 0)]

Close List :  []
 --------------------------
|                          |
|     *        *  *  *  *  |
|     *           G     *  |
|  S     *  *  *  *  *  *  |
|                          |
 --------------------------


Open List :  [(3, 0), (2, 0)]

Close List :  [(3, 1), (4, 0)]
 --------------------------
|                          |
|     *        *  *  *  *  |
|  $  *           G     *  |
|  S     *  *  *  *  *  *  |
|                          |
 --------------------------


Open List :  [(3, 0), (2, 0), (3, 1)]

Close List :  [(1, 0), (4, 0)]
 --------------------------
|                          |
|     *        *  *  *  *  |
|  $  *           G     *  |
|  S  $  *  *  *  *  *  *  |
|                          |
 --------------------------


Open List :  [(3, 0), (2, 0), (3, 1), (1, 0)]

Close List :  [(4, 1), (4, 0)]
 --------------------------
|                          |
|  $  *        *  *  *  *  |
|  $  *           G     *  |
|  S  $  *  *  *  *  *  *  |
|                          |
```

```
| $ *   $ * * * * $|
| $ *   $   G   * $|
| S $ * * * * * * $|
|   $ $ $ $ $ $ $ $|
 --------------------------


Open List :  [(3, 0), (2, 0), (3, 1), (1, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (3, 8), (2, 8), (1, 8), (0, 8), (0, 7), (0, 6), (0
, 5), (0, 4), (0, 3), (1, 3), (2, 3), (2, 4)]

Close List :  [(2, 2), (1, 2), (0, 2), (4, 0), (0, 0), (4, 0)]
 --------------------------
|       $ $ $ $ $ $|
| $ *   $ * * * * $|
| $ *   $ $ G   * $|
| S $ * * * * * * $|
|   $ $ $ $ $ $ $ $|
 --------------------------


Open List :  [(3, 0), (2, 0), (3, 1), (1, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (3, 8), (2, 8), (1, 8), (0, 8), (0, 7), (0, 6), (0
, 5), (0, 4), (0, 3), (1, 3), (2, 3), (2, 4), (2, 5)]

Close List :  [(2, 2), (1, 2), (0, 2), (4, 0), (0, 0), (4, 0)]
 --------------------------
|       $ $ $ $ $ $|
| $ *   $ * * * * $|
| $ *   $ $ G   * $|
| S $ * * * * * * $|
|   $ $ $ $ $ $ $ $|
 --------------------------



Reached goal!!!

Best Route :
 --------------------------
|       $ $ $ $ $ $|
|   *   $ * * * * $|
|   *   $ $ G   * $|
| S $ * * * * * * $|
|   $ $ $ $ $ $ $ $|
 --------------------------
```

4. CITY PROBLEM:

CODE:

```
SOURCE = "Pune"
DESTINATION = "Aurangabad"

DISTANCE_FROM_DESTINATION = {
    "Solapur": 340,
    "Satara": 177,
    "Navi Mumbai": 181,
    "Nashik": 133,
    "Ahmednagar": 180,
    "Aurangabad": 0,
    "Nanded": 451,
    "Latur": 373,
    "Pune": 424,
}

GRAPH = {}

def put_city(city1: str, city2: str, weight: int) -> None:
    global GRAPH
```

```
    if city1 in GRAPH:
        GRAPH[city1].append((city2, weight))
    else:
        GRAPH[city1] = [(city2, weight)]

    if city2 in GRAPH:
        GRAPH[city2].append((city1, weight))
    else:
        GRAPH[city2] = [(city1, weight)]

def generate_map() -> None:
    put_city("Pune", "Navi Mumbai", 106)
    put_city("Pune", "Satara", 112)
    put_city("Pune", "Solapur", 234)
    put_city("Solapur", "Satara", 203)
    put_city("Solapur", "Latur", 104)
    put_city("Nanded", "Latur", 113)
    put_city("Nanded", "Aurangabad", 221)
    put_city("Nashik", "Aurangabad", 159)
    put_city("Nanded", "Ahmednagar", 267)
    put_city("Pune", "Ahmednagar", 120)
    put_city("Nashik", "Pune", 165)
    put_city("Nashik", "Navi Mumbai", 136)


def distance(from_: str, to: str) -> int:
    if from_ == to:
        return 0

    array = GRAPH[from_]
    for name, weight in array:
        if name == to:
            return weight

    return 10 ** 10

class Node:
    def __init__(self, city: str, prev: object, weight: int) -> None:
        self.city = city
        self.prev = prev
        self.weight = weight

    def __repr__(self) -> str:
```

```
        return self.city

    def __eq__(self, other) -> bool:
        if isinstance(other, Node):
            return self.city == other.city
        elif isinstance(other, str):
            return self.city == other
        else:
            return False

def objective(city) -> int:
    return DISTANCE_FROM_DESTINATION[city]

def main() -> None:
    generate_map()
    print("\nStart city : Pune")
    print("\nDestination city : Aurangabad")

    current = Node(SOURCE, None, 0)
    opened = []
    closed = [current]

    while len(closed) > 0:
        print("\nOpened list : ", opened)
        print("\nClosed list : ", closed)

        closed.sort(
                key=lambda node: objective(node.city),
                reverse=True
        )

        current = closed.pop()

        if current not in opened:
            opened.append(current)

        if current == DESTINATION:
            print("\n\n\nReached Destination!!".center(40, ' '))
            break

        print("\nSelected City : ", current)
        print(f"\nThe neighbouring cities of {current} are : ")
        for name, dist in GRAPH[current.city]:
            print(f"{name:>12} with distance {dist}")
```

```
            if name not in opened:
                closed.append(Node(name, current, dist))
        print()

    path = []
    while current:
        path.append(current)
        current = current.prev

    print("\n\nComplete Path : ")
    for city in path[::-1]:
        if city.prev:
            print(f"{city.prev.city} to {city.city} with distance {city.weight}")

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

OUTPUT:

```
Start city : Pune

Destination city : Aurangabad

Opened list :  []

Closed list :  [Pune]

Selected City :  Pune

The neighbouring cities of Pune are :
 Navi Mumbai with distance 106
      Satara with distance 112
     Solapur with distance 234
  Ahmednagar with distance 120
      Nashik with distance 165


Opened list :  [Pune]

Closed list :  [Navi Mumbai, Satara, Solapur, Ahmednagar, Nashik]

Selected City :  Nashik

The neighbouring cities of Nashik are :
  Aurangabad with distance 159
        Pune with distance 165
 Navi Mumbai with distance 136


Opened list :  [Pune, Nashik]

Closed list :  [Solapur, Navi Mumbai, Ahmednagar, Satara, Aurangabad, Navi Mumbai]


Reached Destination!!


Complete Path :
Pune to Nashik with distance 165
Nashik to Aurangabad with distance 159
```

5. 8 PUZZLE:

CODE:

```
GOAL = {
   1: 1, 2: 2, 3: 3,
   4: 8, 5: -1, 6: 4,
   7: 7, 8: 6, 9: 5,
}
t_board = dict[int, int]
class Node:
   def __init__(self, board: dict[int, int], steps: int, prev=None, op=None):
      self.board = board.copy()
      self.steps = steps
      self.prev = prev
      self.operation = op

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

   def __hash__(self) -> int:
      return heuristic(self.board)

   def __repr__(self):
      return str(self.heu())

   def heu(self) -> int:
      return self.steps + heuristic(self.board)


def get_empty_pos(puzzle: t_board) -> int:
   for key in puzzle:
      if puzzle[key] == -1:
         return key


def move_up(puzzle: t_board):
   pos = get_empty_pos(puzzle)
   puzzle[pos], puzzle[pos - 3] = puzzle[pos - 3], puzzle[pos]
   return puzzle


def move_down(puzzle: t_board):
```

```
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos + 3] = puzzle[pos + 3], puzzle[pos]
    return puzzle


def move_right(puzzle: t_board):
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos + 1] = puzzle[pos + 1], puzzle[pos]
    return puzzle


def move_left(puzzle: t_board):
    pos = get_empty_pos(puzzle)
    puzzle[pos], puzzle[pos - 1] = puzzle[pos - 1], puzzle[pos]
    return puzzle


def get_available_operations(puzzle: t_board):
    operations = {
        move_up,
        move_left, move_right,
        move_down,
    }
    empty_pos = get_empty_pos(puzzle)

    if empty_pos in {1, 2, 3}:
        operations.remove(move_up)
    if empty_pos in {7, 8, 9}:
        operations.remove(move_down)
    if empty_pos in {3, 6, 9}:
        operations.remove(move_right)
    if empty_pos in {1, 4, 7}:
        operations.remove(move_left)

    return operations


def get_operation_name(op):
    return {
        move_up: "\nMove Empty Slot UP",
        move_down: "\nMove Empty Slot DOWN",
        move_left: "\nMove Empty Slot LEFT",
        move_right: "\nMove Empty Slot RIGHT",
    }[op]
```

```
def heuristic(puzzle: t_board) -> int:
    score = 0

    for key in puzzle:
        if puzzle[key] != GOAL[key]:
            score += 1

    return score


def safe_get(puzzle: t_board, key: int) -> str:
    if puzzle[key] != -1:
        return puzzle[key]
    return '@'


def display_board(puzzle: dict[int, int]):
    print(
        str(safe_get(puzzle, 1)).center(3, ' ') +
        str(safe_get(puzzle, 2)).center(3, ' ') +
        str(safe_get(puzzle, 3)).center(3, ' ')
    )
    print('-' * 10)
    print(
        str(safe_get(puzzle, 4)).center(3, ' ') +
        str(safe_get(puzzle, 5)).center(3, ' ') +
        str(safe_get(puzzle, 6)).center(3, ' ')
    )
    print('-' * 10)
    print(
        str(safe_get(puzzle, 7)).center(3, ' ') +
        str(safe_get(puzzle, 8)).center(3, ' ') +
        str(safe_get(puzzle, 9)).center(3, ' ')
    )

def start(puzzle) -> Node:
    root = Node(puzzle, 0)
    opened = []
    closed = [root]
    steps = 0
    previous = set()
    best_score = 1000
```

```
    while len(closed) > 0:
        #print("Opened:", opened)
        #print("Closed:", closed)
        steps += 1
        closed.sort(key=lambda n: heuristic(n.board))
        current = closed.pop(0)
        opened.append(current)
        score = heuristic(current.board)

        if score == 0:
            return current
        if current in previous:
            continue
        else:
            previous.add(current)

        if score < best_score:
            best_score = score

        for operation in get_available_operations(current.board):
            child = Node(operation(current.board.copy()),
                         steps, current, operation)
            closed.append(child)

    return current


def main():
    puzzle = {
        1: 1, 2: 3, 3: -1,
        4: 8, 5: 2, 6: 6,
        7: 7, 8: 5, 9: 4,
    }

    print("\nInitial State : ")
    display_board(puzzle)
    print("\nGoal State : ")
    display_board(GOAL)
    print(f"The objective function score of this board is {heuristic(puzzle)}")
    node = start(puzzle)

    path: list[Node] = []
```

```
    while node:
        path.append(node)
        node = node.prev
    path = path[::-1]
    path = path[1:]

    for i, node in enumerate(path):
        print(f"\n\nStep {i + 1} ")
        if node.operation:
            print(get_operation_name(node.operation))

        display_board(node.board)
        print(f"\nObjective Function Score : {heuristic(node.board) + i + 1}")

    if heuristic(path[-1].board) != 0:
        print("\nGoal State not Achieved!")
    else:
        print("\nGoal State Achieved")


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

OUTPUT:

```
Initial State :
 1  3  @
----------
 8  2  6
----------
 7  5  4

Goal State :
 1  2  3
----------
 8  @  4
----------
 7  6  5
The objective function score of this board is 6


Step 1

Move Empty Slot LEFT
 1  @  3
----------
 8  2  6
----------
 7  5  4

Objective Function Score : 6


Step 2

Move Empty Slot DOWN
 1  2  3
----------
 8  @  6
----------
 7  5  4

Objective Function Score : 5
```

```
----------
 8  6  @
----------
 7  5  4

Objective Function Score : 7


Step 4

Move Empty Slot DOWN
 1  2  3
----------
 8  6  4
----------
 7  5  @

Objective Function Score : 7


Step 5

Move Empty Slot LEFT
 1  2  3
----------
 8  6  4
----------
 7  @  5

Objective Function Score : 7


Step 6

Move Empty Slot UP
 1  2  3
----------
 8  @  4
----------
 7  6  5

Objective Function Score : 6

Goal State Achieved
```