Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

Name	Tejas Jadhav
UID No.	2022301006
Class	COMPS A (B batch)
Experiment No.	8

Aim: To use Branch and bound to solve the 15 Puzzle problem

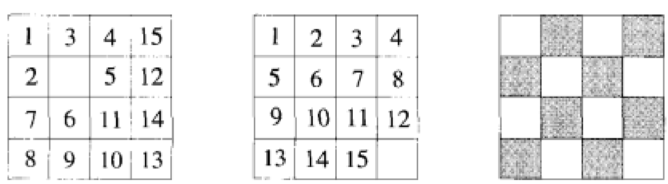
Theory:

- **Branch and Bound Algorithm:**

Branch and bound is a general algorithmic technique used in optimization problems where an exhaustive search is infeasible. The technique divides the search space into smaller subspaces that can be explored efficiently. This method optimizes the search by eliminating partial solutions that cannot possibly be completed to a better solution than the current best solution found so far.

- **15 Puzzle Problem**

The 15 puzzle problem is a well-known sliding puzzle game. The puzzle is played on a 4x4 grid that contains 15 numbered tiles and one empty tile. The goal of the game is to rearrange the tiles to get them in order from 1 to 15, with the empty space in the bottom right corner, in the least number of moves.



(a) An arrangement (b) Goal arrangement (c)

Algorithm:

```
Solve15Puzzle(arr)
    move_count = 0
    misplaced_count = misplaced(arr)
    cost = 0
    prev_move = ' '

    while (misplaced_count != 0)
        move_count++
        cost_up = ∞
        cost_down = ∞
        cost_left = ∞
        cost_right = ∞

        if (prev_move != 'd' AND can_move(arr, 'u'))
            move(arr, 'u')
            cost_up = misplaced(arr) + move_count
            move(arr, 'd')

        if (prev_move != 'u' AND can_move(arr, 'd'))
            move(arr, 'd')
            cost_down = misplaced(arr) + move_count
            move(arr, 'u')

        if (prev_move != 'r' AND can_move(arr, 'l'))
            move(arr, 'l')
            cost_left = misplaced(arr) + move_count
            move(arr, 'r')

        if (prev_move != 'l' AND can_move(arr, 'r'))
            move(arr, 'r')
            cost_right = misplaced(arr) + move_count
            move(arr, 'l')

        if (cost_up < cost_down AND cost_up < cost_left AND cost_up < cost_right
AND can_move(arr, 'u'))
```

move(arr, 'u')
prev_move = 'u'
cost = cost_up
else if (cost_down < cost_left AND cost_down < cost_right AND can_move(arr, 'd'))
move(arr, 'd')
prev_move = 'd'
cost = cost_down
else if (cost_left < cost_right AND can_move(arr, 'l'))
move(arr, 'l')
prev_move = 'l'
cost = cost_left
else
move(arr, 'r')
prev_move = 'r'
cost = cost_right

misplaced_count = misplaced(arr)

return arr

Code:

```
#include <bits/stdc++.h>
using namespace std;


int** create4x4Array() {
    int** arr = new int*[4];
    for (int i = 0; i < 4; i++) {
        arr[i] = new int[4];
    }
    return arr;
}


void input4x4Array(int** arr) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
```

```
            cin >> arr[i][j];
        }
    }
}
void output4x4Array(int** arr) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            cout << setw(3) << arr[i][j] << " ";
        }
        cout << endl;
    }
}

pair<int, int> find_empty_position(int** arr) {
    pair<int, int> pos;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (arr[i][j] == 0) {
                pos.first = i;
                pos.second = j;
                return pos;
            }
        }
    }
    return pos;
}

void swap(int** arr, pair<int, int> pos1, pair<int, int> pos2) {
    int temp = arr[pos1.first][pos1.second];
    arr[pos1.first][pos1.second] = arr[pos2.first][pos2.second];
    arr[pos2.first][pos2.second] = temp;
}

void move_up(int** arr) {
    pair<int, int> pos = find_empty_position(arr);
    if (pos.first == 0) {
        cout << "Can't move up" << endl;
    } else {
        swap(arr, pos, make_pair(pos.first - 1, pos.second));
    }
}
```

```
void move_down(int** arr) {
    pair<int, int> pos = find_empty_position(arr);
    if (pos.first == 3) {
        cout << "Can't move down" << endl;
    } else {
        swap(arr, pos, make_pair(pos.first + 1, pos.second));
    }
}
void move_left(int** arr) {
    pair<int, int> pos = find_empty_position(arr);
    if (pos.second == 0) {
        cout << "Can't move left" << endl;
    } else {
        swap(arr, pos, make_pair(pos.first, pos.second - 1));
    }
}
void move_right(int** arr) {
    pair<int, int> pos = find_empty_position(arr);
    if (pos.second == 3) {
        cout << "Can't move right" << endl;
    } else {
        swap(arr, pos, make_pair(pos.first, pos.second + 1));
    }
}

void move(int** arr, char c) {
    switch (c) {
        case 'u':
            move_up(arr);
            break;
        case 'd':
            move_down(arr);
            break;
        case 'l':
            move_left(arr);
            break;
        case 'r':
            move_right(arr);
            break;
        default:
```

```
            cout << "Invalid move" << endl;
    }
}

int misplaced(int** arr) {
    int count = 0;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (arr[i][j] != (4 * i + j + 1) % 16) {
                count++;
            }
        }
    }
    return count;
}

bool can_move(int** arr, char c) {
    pair<int, int> pos = find_empty_position(arr);
    switch (c) {
        case 'u':
            return pos.first != 0;
        case 'd':
            return pos.first != 3;
        case 'l':
            return pos.second != 0;
        case 'r':
            return pos.second != 3;
        default:
            return false;
    }
}

void solve15puzzle(int** arr) {

    int move_count = 0;
    int misplaced_count = misplaced(arr);
    int cost = 0;

    char prev_move = ' ';

    while (misplaced_count != 0) {
```

```
        move_count++;
        int cost_up, cost_down, cost_left, cost_right;

        if (prev_move != 'd' && can_move(arr, 'u')) {
            move(arr, 'u');
            cost_up = misplaced(arr) + move_count;
            move(arr, 'd');
        }
        if (prev_move != 'u' && can_move(arr, 'd')) {
            move(arr, 'd');
            cost_down = misplaced(arr) + move_count;
            move(arr, 'u');
        }

        if (prev_move != 'r' && can_move(arr, 'l')) {
            move(arr, 'l');
            cost_left = misplaced(arr) + move_count;
            move(arr, 'r');
        }

        if (prev_move != 'l' && can_move(arr, 'r')) {
            move(arr, 'r');
            cost_right = misplaced(arr) + move_count;
            move(arr, 'l');
        }

        if (cost_up <= cost_down && cost_up <= cost_left && cost_up <=
cost_right) {
            move(arr, 'u');
            prev_move = 'u';
            cost = cost_up;
        } else if (cost_down <= cost_left && cost_down <= cost_right) {
            move(arr, 'd');
            prev_move = 'd';
            cost = cost_down;
        } else if (cost_left <= cost_right) {
            move(arr, 'l');
            prev_move = 'l';
            cost = cost_left;
        } else {
            move(arr, 'r');
```

```
            prev_move = 'r';
            cost = cost_right;
        }
        cout << endl <<  move_count  << ") " ;
        cout << "Move: " << prev_move ;
        cout << "  Cost: " << cost << endl;

        output4x4Array(arr);
        misplaced_count = misplaced(arr);
        cout << endl;
    }
}

int main() {
    int** arr = create4x4Array();
    cout << "Enter inital state of 15 puzzle: " << endl;
    input4x4Array(arr);
    cout << "\n\nInitial state of 15 puzzle: " << endl;
    output4x4Array(arr);
    solve15puzzle(arr);
    return 0;
}
```

Output:

```
PS D:\Tejas\clg\daa\Experiment 08\code> ./a
Enter inital state of 15 puzzle:
1 2 3 4
5 6 0 8
9 10 7 11
13 14 15 12


Initial state of 15 puzzle:
  1   2   3   4
  5   6   0   8
  9  10   7  11
 13  14  15  12

Costs: 6 4 6 6

1) Move: d  Cost: 4
  1   2   3   4
  5   6   7   8
  9  10   0  11
 13  14  15  12


Costs: 2147483647 6 6 4

2) Move: r  Cost: 4
  1   2   3   4
  5   6   7   8
  9  10  11   0
 13  14  15  12


Costs: 6 3 2147483647 2147483647

3) Move: d  Cost: 3
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15   0

PS D:\Tejas\clg\daa\Experiment 08\code>
```

Conclusion:

In conclusion, the branch and bound algorithm is an effective technique for solving optimization problems, including the 15 puzzle problem. By dividing the search space into smaller subspaces and eliminating partial solutions that cannot lead to a better solution, this algorithm reduces the computational complexity of the problem, resulting in a more efficient and optimal solution.